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 provides secure MPC 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 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.

1 Introduction

Secure multi-party computation (MPC; [25]) 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 could allow medical research institutions to jointly train better diagnostic models without having to share their sensitive patient data [11] or allow social scientists to analyze gender wage gap statistics without companies having to share sensitive salary data [17]. 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 [15, 23, 24]. 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 tecniques in machine learning, we are developing CRYPTEN: a software framework that aims to make modern secure MPC techniques accessible to machine-learning researchers and developers without a background in cryptography. Specifically, CRYPTEN provides a comprehensive tensor-computation library in which all computations are performed via secure MPC. On top of this tensor library, it provides automatic differentiation and a modular neural-network package. CRYPTEN's API closely follows the API of the popular PyTorch framework for machine learning [21], which makes it easy to use for machine-learning practitioners. CRYPTEN assumes an *honest-but-curious* threat model and works for an arbitrary number of parties. The paper presents: (1) an overview of CRYPTEN's design principles, (2) a description of the secure MPC protocols implemented in CRYPTEN, (3) examples of how CRYPTEN can be used to implement modern machine-learning models, and (4) a roadmap for the further development of CRYPTEN.

Preprint. Under review.

2 Design Principles

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

Machine-learning first. CRYPTEN has a general purpose, machine-learning first API design. Many other secure MPC frameworks (see [13] for an overview) 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 [21], implements reverse-mode automatic differentiation, and provides a modular neural-network package with corresponding learning routines. We aim to allow developers to switch their code from PyTorch to CRYPTEN by changing a single Python `import` statement.

Eager execution. CRYPTEN adopts an imperative programming model. This is different from most existing MPC frameworks, which implement compilers for their own domain-specific languages. While compiler approaches have potential performance benefits, they slow down the development cycle, make debugging harder, and complicate integration with existing software. Instead, CRYPTEN follows the recent trend in machine learning away from graph compilers [1] to frameworks that perform eager execution of computation graphs [21]. This allows for easy code tracing and debugging. 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 specifically developed for high-performance distributed computing.

3 Overview of Secure MPC Protocols

To facilitate secure computations, CRYPTEN implements arithmetic secret sharing [9] and binary secret sharing [12], as well as logic that converts between these two types of secret sharing [10]. Arithmetic secret sharing is particular well-suited for operations that are common on modern machine-learning models, such as matrix multiplications and convolutions. Binary secret sharing is required for evaluating certain other common functions, including the rectified linear units and argmaxes.

3.1 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 [7] 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: $x = \lfloor Bx_R \rceil$, 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 Ripple-carry adder in $\log_2(|\mathcal{P}|)\log_2(L)$ communication rounds [6, 8].

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 *trusted third party* (TTP) but we are also working on an implementation that generates them off-line via oblivious transfer [16]. The parties use $\langle r^{(b)}\rangle$ to mask $\langle x\rangle^{(b)}$ and reveal the resulting masked bit $z^{(b)}$. Subsequently, they compute $\left[\langle x\rangle^{(b)}\right] = \left[r^{(b)}\right] + z^{(b)} - 2\left[r^{(b)}\right] z^{(b)}$.

3.2 Secure Computation

Arithmetic and binary secret shares have homomorphic properties that can be used to implement secure computation. 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 [4], $([a], [b], [c])$ with $c = ab$. The Beaver triples are currently provided by the TTP but we are developing protocols in which the parties generate Beaver triples via additive homomorphic encryption [20] or oblivious transfer [16]. The parties compute $[\epsilon] = [x] - [a]$ and $[\delta] = [y] - [b]$, and decrypt ϵ and δ without information leakage due to the masking. Finally, 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 [14], 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 $[c\ ?\ x : y] = [c][x] + (1 - [c])[y]$.

4 Examples of CRYPTEN Usage

Figure 1 shows an example of how PyTorch tensors are secret-shared and revealed in CRYPTEN. 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), this communication acts as a synchronization point. In the example, the input tensor for the creation of the arithmetic secret share is provided by all parties. However, the input tensor will often be provided by one specific party in real-world applications of secure MPC. This can be achieved by using the optional `src` argument in the `cryptensor` constructor: for example, `crypten.cryptensor(x, src=0)` will secret-share the PyTorch tensor `x` that is owned by the party with rank 0 with the other parties.[1]

```
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)

# 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)
xy_enc = x_enc + y_enc
xy_dec = xy_enc.get_plain_text()
assert torch.all_close(xy_dec, x + y)
```

Figure 1: Example of secret-sharing tensors, revealing tensors, and private addition in CRYPTEN.

Figure 2 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 it computations, and it also makes debugging CRYPTEN code easier.

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

Figure 3 demonstrates how existing PyTorch models can be imported into CRYPTEN. Models are imported via ONNX, which makes CRYPTEN compatible with many other deep-learning frameworks (TensorFlow is already supported). As the example illustrates, performing private inference using a ResNet-18 model is straightforward in CRYPTEN. The example in the figure also demonstrates CRYPTEN's GPU support. One caveat is that, at present, all parties must use the same device (CPU or GPU) for their computations. Another caveat is that CRYPTEN only supports a subset of the operations that PyTorch provides. Due to its use of secure MPC, CRYPTEN also has substantial computational overhead compared to PyTorch.

5 Roadmap

Although CRYPTEN is quickly maturing, it still has three important directions for development.

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.

Differential privacy implementations may be required in a range of 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. The implementation of differentially private mechanisms in secure MPC has received little attention ([22] is a notable exception) and is non-trivial. In particular, the *honest-but-curious* threat model is insufficient for some implementations of differential privacy, and one must ensure that the implementations are not amenable to floating-point attacks [18]. We plan the discrete Laplace mechanism [5] in CRYPTEN in a way that is not susceptible to such attacks and does not require *active security*. We also plan support for active security, and to use this support to implement the discrete Gaussian mechanism [3, 5].

End-to-end privacy requires seamless integration between data-processing frameworks, such as secure SQL implementations [2], 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.

Acknowledgments. We thank Joe Spisak and the entire PyTorch team for their support of CRYPTEN.

```
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 2: Example using neural networks and automatic differentiation in CRYPTEN.

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

# 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 3: Private inference on secret-shared images using a secret-shared ResNet-18 model on GPU.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Usenix*, 2016.

[2] 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.

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

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

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

[6] 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.

[7] R. Cramer, I. Damgard, 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.

[8] 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.

[9] 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`.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] A. Paszke, S. Gross, S. Chintala, and G. Chanan. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, 2017.

[22] M. Pettai and P. Laud. Combining differential privacy and secure multiparty computation. In *Proceedings of the Annual Computer Security Applications Conference*, pages 421–430, 2015.

[23] 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.

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

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

A Detailed Description of Secure MPC Protocols

A.1 Secret Sharing

CRYPTEN uses two different types of secret sharing: (1) arithmetic secret sharing [9]; and (2) binary secret sharing [12]. Below, we describe the secret sharing methods for single values x but they can trivially be extended to real-valued vectors $\mathbf{x}$.

A.1.1 Arithmetic Secret Sharing

CRYPTEN uses arithmetic secret sharing to perform 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 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 $B = 2^L$ to encode floating-point numbers, we must scale down by a factor 2^L after every multiplication. We do this using the public division protocol described under "truncation" 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 $[y]$. That is, 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 [4], 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 where the triple (a, b, c) satisfies $c = f(a, b)$, we use also use this process in CRYPTEN to compute matrix multiplication and convolution.

Square. To compute the square $[x^2]$, the parties use a Beaver pair $([a], [b])$ such that $b=a^2$. As before, the parties then 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 simply 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 $P(\theta_x \neq 0) = \frac{x}{Q}$ in the two-party case. Many prior MPC implementations specialize to the $|\mathcal{P}| = 2$ case and rely on this probability being negligible [19, 23, 24]. 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 do so, 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 modulus.

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: $[\theta_x] \leftarrow \theta_z + [\beta_{xr}] - [\theta_r] - [\eta_{xr}]$

- 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 $[\theta_x]$. 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 whether the number of parties is two or larger, *i.e.*, regardless of the number of parties $P(\eta_{xr} \neq 0) = \frac{x}{Q}$. In practice, we 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 multiply two real-values, $\hat{x}$ and $\hat{y}$, the result will be encoded as $B^2\hat{x}\hat{y}$ which has probability $\frac{B^2\hat{x}\hat{y}}{Q}$ of producing an error. This probability can be reduced by increasing Q or reducing the precision parameter, B.

A.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 existing bitwise operations on integer-type variables 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, subjecting all but very simple circuits to be inefficient to compute via binary secret sharing.

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 p XOR their shares of $\langle x \rangle$ and $\langle y \rangle$. That is, efach party $p \in \mathcal{P}$ computes $\langle z \rangle_p \leftarrow \langle x \rangle_p \oplus \langle y \rangle_p$.

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: $[b] \leftarrow [r] + z - 2[r]z$.

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 from a uniform Bernoulli distribution. $\langle x \rangle \otimes \langle y \rangle$ can now be evaluated by computing $\langle c \rangle \oplus (\epsilon \otimes \langle b \rangle) \oplus (\langle a \rangle \otimes \delta) \oplus (\epsilon \otimes \delta)$, where revealing ϵ and δ requires a round of communication among all parties. Correctness follows from the same logic as multiplication in arithmetic secret sharing.

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, $\langle y \rangle_p = \text{shift}(\langle x \rangle_p, k)$.

A.1.3 Converting Between Secret-Sharing Types

Machine Learning applications involve both functions that are easier to compute on arithmetic secret shares (*e.g.*, matrix multiplication) and functions that are easier to implement via on binary secret shares (*e.g.*, argmax) using binary circuits. Therefore, we use both types of secret sharing and convert between the two types using the techniques proposed in [10].

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 Ripple-carry adder circuit can be evaluated in $\log_2(|\mathcal{P}|) \log_2(L)$ rounds [6, 8].

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, where $[r]$ and $\langle r \rangle$ represent arithmetic and binary secret-shares of the same one-bit value r, respectively. 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 1 (*i.e.*, to reveal z).

A.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 equality 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 MPC, 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.

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

A.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 using MPC. Each of these approximations has a specific domain over which the approximation converges well. One can modify the domain of convergence for certain functions by noting 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 to allow users to make protocol-specific optimizations, like using custom initializations to improve convergence rates for iterative methods given a known input domain.

A.2.1 Exponential, Sine, and Cosine

There are many well-known polynomial approximations for the exponential function, for example the Taylor Series approximation, $e^x = \sum_{n=0}^{\infty} \frac{1}{n!} x^n$. However, because exponentials grow much faster than polynomials, the degree of the polynomial we need to approximate the exponential function increases exponentially as our domain size increases. We use the following limit approximation to counteract this since we can use repeated squaring to generate very high order polynomials quickly:

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

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

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

A.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 Netwon-Raphson convergence criterion, which for the above method is $\frac{1}{2x} < y_0 < \frac{2}{x}$. By default, CRYPTEN uses the function:

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

as an initialization to provide a large domain of convergence. 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 appropriate singular values that meet the convergence criteria.)

A.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 in MPC. Instead, we use the Newton-Raphson update formula for inverse square root since it is much more efficient:

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

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

A.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 $\ln(x)$ or by manipulating the Taylor Series expansion of $\ln(1 - x)$:

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

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 rather than higher numbers of iterations. By default, CRYPTEN uses a polynomial of degree 8, 3 iterations, and the initialization $y_0 = \frac{x}{120} - 20e^{-2x-1} + 3$, which allows good convergence over 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 $x^y = e^{y \ln(x)}$.

A.2.5 Sigmoid and Tanh

We have explored several methods for computing logistic functions in MPC, including direct computation, rational approximations, and Chebyshev polynomial approximations. We have found that with optimizations, the direct computation is the most efficient. This method uses the exponential and reciprocal functions to compute:

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

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

A.3 Random Sampling

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

A.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. These bits can be generated by having each party randomly generate 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. This means no party can gain any information about the resulting bit from its own binary share of the bit. These bits can then be converted to an arithmetic secret-share $[u]$.

A.3.2 Bernoulli Sampling

To compute a Bernoulli random variable with arbitrary mean $[b] \sim Bern(p)$, we can 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 will be quantized to the nearest 2^{-L}, as we would expect if p were encoded using the same fixed-point encoder.

A.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 normal random variables $([x_1], [x_2])$ with 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. The result of these equations will be two independent random variables with distribution $[x] \sim \mathcal{N}(0, 1)$. To extend these to $[y] \sim \mathcal{N}(\mu, \sigma^2)$, we compute $[y] = \sigma[x] + \mu$.

A.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 can be 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 $[b] \sim Bern(0.5)$ by computing $[y] = (2[b] - 1)[x]$.

A.3.5 Weighted random sampling

To produce a weighted random sample of an input $[x]$ with weights given by $[w]$, we first generate a uniform random sample in $[0, \sum[w]]$ by drawing a random sample $[u] \sim Uniform(0, 1)$ and multiplying $[r] = [u][\sum[w]]$. Care should be taken to avoid precision issues caused by generating $[u]$ in fixed-point with finite precision. We then compute the cumulative sum $[c]$ of $[w]$ and compare to our random value $[m_i] = [c > [r]]$. This us a mask whose entries are all zeros below some index i and all ones above i. To get a one-hot mask, we shift all elements of $[m_i]$ to get $[m_{i+1}]$ and subtract: $[m] = [m_i] - [m_{i+1}]$. We can then sample from our input $[x]$ by multiplying by our mask $[y] = [x][m]$.