Moose documentation

June 3, 2022

Contents

1 Introduction

This paper details the cryptographic techniques and protocols used by Cape Privacy's encrypted learning solution. We expect the reader to have some familiarity with secure multi-party computation, linear algebra, and ring arithmetic; as a good starting point we recommend [EKR17]. We also omit detailed explanations of machine learning and the models we support, instead referring the interested reader to [DFC20].

The main purpose of this paper is to describe some of the design decisions that went into Moose and the current MPC protocols that are implemented in it.

2 Fixed-point Arithmetic

Our cryptographic protocols natively operate on ring elements, and we must use ring arithmetic to emulate other data types and computations. For this reason we use fixed-point arithmetic instead of floating point arithmetic, which results in more efficient computations. In this section we briefly outline fixed-point arithmetic and how to emulate it with ring arithmetic in $\mathbb{Z}_{2^k}$, which in turn can be emulated with our cryptographic protocols and implemented using integer instructions. For efficiency reasons, we only consider $k = 64$ and $k = 128$, since these give us modulus reductions for free when using wrapping instructions.

2.1 Fixed-point Numbers

We let $\texttt{fixed}(k, f)$ be the subset of the real numbers

$$\{x \in \mathbb{R} : x = \bar{x} \cdot 2^{-f}, \bar{x} \in \mathbb{Z}_{2^k}\}$$

that have a fixed-point representation in $\mathbb{Z}_{2^k}$ using f bits fractional precision. Here, $\mathbb{Z}_{2^k}$ is considered as the k bit signed integers $\{\bar{x} : -2^{k-1} \leq \bar{x} < 2^{k-1}\}$.

2.2 Encoding and Decoding

To encode a number $x \in \mathbb{R}$ into its fixed-point representation $\bar{x} \in \mathbb{Z}_{2^k}$, we compute $\mathsf{Encode}(x, f) = \lfloor x \cdot 2^f \rfloor$ where the result of flooring is cast as an integer. To decode a number $\bar{x}$, we compute $\mathsf{Decode}(\bar{x}, f) = x \cdot 2^{-f}$ where $\bar{x}$ is first cast as a floating point.

2.3 Addition

To add two fixed-point numbers $\bar{x}$ and $\bar{y}$ where $x, y \in \texttt{fixed}(k, f)$, we can simply compute $\bar{z} = \bar{x} + \bar{y}$ using $\mathbb{Z}_{2^k}$ arithmetic. Note that $\mathsf{Decode}(\bar{z}) = x + y$ if and only if $x + y \in \texttt{fixed}(k, f)$.

2.4 Multiplication

To multiply two fixed-point numbers $\bar{x}$ and $\bar{y}$ where $x, y \in \texttt{fixed}(k, f)$, we can first compute $\bar{z} = \bar{x} \cdot \bar{y}$ using $\mathbb{Z}_{2^k}$ arithmetic. However, in general $z \in \texttt{fixed}(k, 2f)$ and we see that the fractional precision of the numbers grow by f bits with each multiplication. To circumvent this we introduce the $\mathsf{Trunc}(\bar{z})$ operation which computes $\bar{z}/2^f$ over the integers, which brings us back to $z \in \texttt{fixed}(k, f)$. In the main body the Trunc operation is replaced by a probabilistic variant $\mathsf{TruncPR}$ that allows for an error in the least significant bit in exchange for better performance.

2.5 Inverse and Division

Given a fixed-point $\bar{x}$ we can compute its inverse $1/x$ over the reals by computing the integer division of $\bar{1} \cdot 2^f = 1 \cdot 2^{2f}$ and $\bar{x}$, so that the result $\bar{y} = \bar{1} \cdot 2^f / \bar{x} = y \cdot 2^f$ for some $y \in \texttt{fixed}(k, f)$. Likewise, the division of $\bar{x}$ by $\bar{y}$ can be computed as the integer division of $\bar{x} \cdot 2^f$ by $\bar{y}$. Note that no truncation is needed.

In order to compute N/D using fixed-point arithmetic only relying on additions and multiplications there are two main lines of thought: Newton-Raphson method or Goldschmidt method.

They both start with finding a good approximation of the reciprocal $D' = 1/D$ where $D' = [0.5, 1]$ in order to find a function $f(X) = 1/X - D = 0$. The Newton-Raphson iteration is given by $X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} = X_i(2 - DX_i)$. The most straightforward choice would be to use Newton-Raphson method for finding a/b described below.

The Goldschmidt method starts with the same reciprocal approximation like Newton-Raphson. However, instead of applying the result of each iteration on the denomitor, one iteration has an effect on the nominator

as well. Once an approximation $D_{-1} \approx 1/D$ is found then one Goldschmidt iteration is the following:

$$\begin{aligned} X_i &= 2 - D_{i-1} \\ D_i &= X_i \cdot D_{i-1} \\ N_i &= X_i \cdot N_{i-1} \end{aligned}$$

where $N_0 = N$ and $D_0 = D$.

2.5.1 NewtonDivFloat(N, D):

1. $D' \leftarrow D$.
2. $p = 1$.
3. While $D' > 1$:
   (a) $D' \leftarrow D'/2$.
   (b) $p \leftarrow p \cdot 2$.
4. $X \leftarrow 48/17 - 32/17 \cdot D'$. (initial approximation)
5. For $i \in [1, \theta]$:
   (a) $X \leftarrow X \cdot (2 - D'X)$.
6. Return $N/p \cdot x$.

3 Computational Model

In this section we describe our computational model for protocols, which is loosely based on the data-flow paradigm [ABC+16] and a simplified UC model [CCL15]. Concretely, protocols are expressed as graphs where nodes represent operations to be performed by a specific party, and edges present values "flowing" between operations.

The main motivation for using the data-flow paradigm is that it leads to a very natural concurrent execution model, where in the extreme we can see each node as being executed by a separate task (e.g. green thread or actor). We take full advantage of this in the runtime which is based on the asynchronous execution paradigm. The reason for basing our computational model on the UC model is that it is a well-known paradigm for ensuring security under concurrent composition.

3.1 Sessions

Every execution of a graph is performed under a unique session id sid used to identify values and ensure isolation when running protocols concurrently. As we shall see in more detail later, session ids are for instance used to non-interactively derive nonces and sample correlated randomness by the secret sharing schemes.

Session ids must be unique and of fixed length for security reasons, but can otherwise safely be chosen by an untrusted coordinator, for instance by sampling a random string. To satisfy the security requirements, each party maintains a list of previous session ids in which it has engaged, and refuses to re-run any computation using those ids; this prevents for instance replay and selective failure attacks. It additionally checks that session ids have the correct length.

3.2 Sub-protocols

We allow graphs to call sub-graphs similar to calling sub-routines. When doing so, the sub-graph is executed under a sub-session id sid' derived from sid and an activation key $\mathsf{ac\_key}$ statically related to the call site:

$$\mathsf{sid}' = \mathsf{h}(\mathsf{sid} \| \mathsf{ac\_key})$$

with h being a secure hash function and $\|$ denoting string concatenation. For security we require sid to have fixed length ℓ. Calling a sub-graph is done through Enter and Exit operations that are implicitly linked to Input and Output operations.

3.3 Communication

Transmission of values between parties is done using Send and Receive nodes where each pair is linked together by a static $\mathsf{rdv\_key}$ attribute. Together with the session id, this allows us to uniquely identify all values by tagging them with $(\mathsf{sid}, \mathsf{rdv\_key})$ during transmission.

3.4 Inlining

In the presentation given in this paper we make heavy use of calling sub-protocols, yet for performance reasons it may be interesting to inline sub-graphs. To do so securely, special attention must be paid to certain node attributes that control uniqueness.

As an example, the $\mathsf{rdv\_key}$ attribute of Send and Receive nodes in the graph being inlined must be updated to $\mathsf{h}(\mathsf{ac\_key} \| \mathsf{rdv\_key})$ where $\mathsf{ac\_key}$ is the activation key of the Enter and Exit nodes being replaced, and $\mathsf{rdv\_key}$ in the graph being inlined into must be updated to $\mathsf{h}(\mathsf{rdv\_key})$. Other examples are the $\mathsf{sync\_key}$ attribute of DeriveSeed operations and the $\mathsf{ac\_key}$ attribute of Enter and Exit operations. For this to be secure we require all $\mathsf{ac\_key}$, $\mathsf{rdv\_key}$, and $\mathsf{sync\_key}$ to be of fixed length ℓ.

4 Replicated Protocols

In this section we describe our protocols for performing encrypted computations using the three-party replicated secret sharing scheme. Much of this work follows the lines of [AFL+16] with some optimizations derived from the fact that we focus on tensor computations. All protocols presented here operate in the honest-but-curious security model, meaning players are assumed to follow the protocols but may try to learn additional information from the messages they receive.

One reason for choosing [AFL+16] as our foundation is that it is currently the fastest protocol for achieving semi-honest three-party multiplications in the preprocessing model using ring arithmetic. Moreover, its ring variant can be efficiently upgraded to malicious security by (roughly) just repeating all procedures twice and use a "zero check" as described in [DEK20a].

Finally, we also chose this line of work due to its efficiency at computing dot products by communicating a number of ring elements independent in the size of input tensors, and other optimizations made possible by being in the three-party setting.

4.1 Notation

The protocols perform computations in rings R of form $\mathbb{Z}_{2^k}$, concretely $\mathbb{Z}_{2^{64}}$ and $\mathbb{Z}_{2^{128}}$, which allows us to emulate fixed-point arithmetic as needed for the linear regression use case. Throughout this section we use

$$\langle \mathbf{x} \rangle_R^{\mathsf{rep}} = \big((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\big)$$

to denote a replicated value consisting of shares $\mathbf{x}_1 = \mathbf{x}_1^1 = \mathbf{x}_1^3$, $\mathbf{x}_2 = \mathbf{x}_2^2 = \mathbf{x}_2^1$, and $\mathbf{x}_3 = \mathbf{x}_3^3 = \mathbf{x}_3^2$ over ring R such that $\mathbf{x} = \sum_{i=1}^3 \mathbf{x}_i \in \mathbb{Z}_{2^k}$. It will typically be the case that P_i holds $\mathbf{x}_i^i$ and $\mathbf{x}_{i+1}^i$. To ease the notation we let indices wrap around such that for instance $P_{3+1} = P_1$, and we occasionally write $\langle \mathbf{x} \rangle_{\mathbb{Z}_{2^k}}^{\mathsf{rep}}$ as $\langle \mathbf{x} \rangle_{2^k}$ or simply $\langle \mathbf{x} \rangle$ when the rest is clear from the context.

4.2 Setup

Some of the replicated protocols rely on PRF keys produced during an initial setup phase as described by the Setup protocol in Figure 4.2.1. Note that keys are distributed the same way as replicated shares so that P_i ends up knowing both k_i and k_{i+1}. Generation of individual keys by GenPrfKey is implemented by sampling 128 random bits using [Lib21].

4.2.1 $\mathsf{Setup}_{[P_1,P_2,P_3]}$

Replicated setup protocol.

1. On P_i for $i \in [3]$:
   (a) $k_i^i \leftarrow \mathsf{GenPrfKey}()$.
   (b) Send k_i^i to P_{i-1}.
   (c) Receive k_{i+1}^i from P_{i+1}.
2. Return $\mathbf{k} = \big((k_1^1, k_2^1), (k_2^2, k_3^2), (k_3^3, k_1^3)\big)$.

4.3 Addition and Subtraction

To compute the addition of two replicated tensors $\langle \mathbf{z} \rangle = \langle \mathbf{x} + \mathbf{y} \rangle$ the parties simply add their local shares of $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$ as described in protocol RepAdd in Figure 4.3.1. Subtraction as shown in Figure 4.3.2 is almost identical, with the parties simply subtracting their shares locally instead of adding. Correctness and security follow from [ABF+16].

4.3.1 $\mathsf{RepAdd}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle, \langle \mathbf{y} \rangle)$

Replicated addition protocol.

1. Let $\big((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\big) = \langle \mathbf{x} \rangle$.
2. Let $\big((\mathbf{y}_1^1, \mathbf{y}_2^1), (\mathbf{y}_2^2, \mathbf{y}_3^2), (\mathbf{y}_3^3, \mathbf{y}_1^3)\big) = \langle \mathbf{y} \rangle$.
3. On P_i for $i \in [3]$:
   (a) $\mathbf{z}_i^i \leftarrow \mathbf{x}_i^i + \mathbf{y}_i^i$
   (b) $\mathbf{z}_{i+1}^i \leftarrow \mathbf{x}_{i+1}^i + \mathbf{y}_{i+1}^i$
4. Return $\langle \mathbf{z} \rangle = \big((\mathbf{z}_1^1, \mathbf{z}_2^1), (\mathbf{z}_2^2, \mathbf{z}_3^2), (\mathbf{z}_3^3, \mathbf{z}_1^3)\big)$.

4.3.2 $\mathsf{RepSub}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle, \langle \mathbf{y} \rangle)$

Replicated subtraction protocol.

1. Let $\big((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\big) = \langle \mathbf{x} \rangle$.
2. Let $\big((\mathbf{y}_1^1, \mathbf{y}_2^1), (\mathbf{y}_2^2, \mathbf{y}_3^2), (\mathbf{y}_3^3, \mathbf{y}_1^3)\big) = \langle \mathbf{y} \rangle$.
3. On P_i for $i \in [3]$:
   (a) $\mathbf{z}_i^i \leftarrow \mathbf{x}_i^i - \mathbf{y}_i^i$
   (b) $\mathbf{z}_{i+1}^i \leftarrow \mathbf{x}_{i+1}^i - \mathbf{y}_{i+1}^i$
4. Return $\langle \mathbf{z} \rangle = \big((\mathbf{z}_1^1, \mathbf{z}_2^1), (\mathbf{z}_2^2, \mathbf{z}_3^2), (\mathbf{z}_3^3, \mathbf{z}_1^3)\big)$.

4.4 Multiplication

Protocol RepMul in Figure 4.4.1 is used to compute the product of two replicated tensors $\langle \mathbf{z} \rangle = \langle \mathbf{x} \cdot \mathbf{y} \rangle$. It follows the multiplication protocol of [AFL+16] with some small computational optimizations in the underlying $\mathsf{ZeroShare}$ protocol in Figure 4.4.2, and inherit their correctness and security proofs.

As for the $\mathsf{ZeroShare}$ protocol, note that the $\mathsf{DeriveSeed}$ operation is parameterized by an explicit $\mathsf{sync\_key}$ attribute allowing P_i and P_{i+1} to generate the same seeds non-interactively, i.e. $\mathsf{seed}_1^1 = \mathsf{seed}_1^3 \neq \mathsf{seed}_2^1 = \mathsf{seed}_2^2 \neq \mathsf{seed}_3^2 = \mathsf{seed}_3^3$. The operation is implemented as $\mathsf{PRF}(k, \mathsf{sid} \| \mathsf{sync\_key})$, which in turn is implemented using the keyed hash function from [Bla22] (BLAKE3) with an output truncated to 128 bits; This ensures that all seeds are unique across sessions when sid and $\mathsf{sync\_key}$ are of fixed length ℓ. Finally, the $\mathsf{SampleUniformSeeded}$ operation is implemented by running AES as a PRNG using the seed as the key and encrypting plaintexts $0, 1, ...$ in ECB mode, as described by Guo et al [GKWY20].

4.4.1 $\mathsf{RepMul}_{[P_1,P_2,P_3]}(R; \mathbf{k}, \langle \mathbf{x} \rangle_R, \langle \mathbf{y} \rangle_R)$:

Replicated multiplication protocol.

1. Let $\big((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\big) = \langle \mathbf{x} \rangle_R$.
2. Let $\big((\mathbf{y}_1^1, \mathbf{y}_2^1), (\mathbf{y}_2^2, \mathbf{y}_3^2), (\mathbf{y}_3^3, \mathbf{y}_1^3)\big) = \langle \mathbf{y} \rangle_R$.
3. On P_i for $i \in [3]$:
   (a) $\mathbf{v}_i^i \leftarrow \mathbf{x}_i^i \cdot \mathbf{y}_i^i + \mathbf{x}_i^i \cdot \mathbf{y}_{i+1}^i + \mathbf{x}_{i+1}^i \cdot \mathbf{y}_i^i$.
   (b) $\mathsf{shape}_i \leftarrow \mathsf{Shape}(\mathbf{v}_i^i)$.
4. $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \boldsymbol{\alpha}_3 \leftarrow \mathsf{ZeroShare}_{[P_1,P_2,P_3]}\big(R; \mathbf{k}, \mathsf{shape}_1, \mathsf{shape}_2, \mathsf{shape}_3\big)$

5. On P_i for $i \in [3]$:
   (a) $\mathbf{z}_i^i \leftarrow \mathbf{v}_i^i + \boldsymbol{\alpha}_i$.
   (b) Send $\mathbf{z}_i^i$ to P_{i-1}.
   (c) Receive $\mathbf{z}_{i+1}^i$ from P_{i+1}.
6. Return $\langle \mathbf{z} \rangle_R = \big((\mathbf{z}_1^1, \mathbf{z}_2^1), (\mathbf{z}_2^2, \mathbf{z}_3^2), (\mathbf{z}_3^3, \mathbf{z}_1^3)\big)$.

4.4.2 $\mathsf{ZeroShare}_{[P_1,P_2,P_3]}(R; \mathbf{k}, \mathsf{shape}_1, \mathsf{shape}_2, \mathsf{shape}_3)$:

Zero share protocol.

1. Let $\big((k_1^1, k_2^1), (k_2^2, k_3^2), (k_3^3, k_1^3)\big) = \mathbf{k}$ be PRF keys generated during setup.
2. Let $\mathsf{sync\_key}_1, \mathsf{sync\_key}_2, \mathsf{sync\_key}_3$ be distinct values.
3. On P_i for $i \in [3]$:
   (a) $\mathsf{seed}_i^i \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_i; k_i^i)$.
   (b) $\mathsf{seed}_{i+1}^i \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_{i+1}; k_{i+1}^i)$.
   (c) $\boldsymbol{\alpha}_i \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{seed}_i^i, \mathsf{shape}_i) - \mathsf{SampleUniformSeeded}(R; \mathsf{seed}_{i+1}^i, \mathsf{shape}_i)$.
4. Return $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \boldsymbol{\alpha}_3)$.

4.5 Sum and Mean

Similarly to addition, we build a summation protocol (Figure 4.5.1) to compute the sum along an axis of a replicated tensor by simply computing the sum along the axis of the individual shares. This can be extended to a protocol for computing the mean by subsequently performing a (public) multiplication with the fixed-point inverse of the size of the axis n.

4.5.1 $\mathsf{RepSum}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle, \mathsf{axis})$:

Replicated sum over an axis.

1. Let $\big((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\big) = \langle \mathbf{x} \rangle$.
2. On P_i for $i \in [3]$:
   (a) $\mathbf{z}_i^i \leftarrow \mathsf{Sum}(\mathbf{x}_i^i, \mathsf{axis})$
   (b) $\mathbf{z}_{i+1}^i \leftarrow \mathsf{Sum}(\mathbf{x}_{i+1}^i, \mathsf{axis})$
3. Return $\langle \mathbf{z} \rangle = \big((\mathbf{z}_1^1, \mathbf{z}_2^1), (\mathbf{z}_2^2, \mathbf{z}_3^2), (\mathbf{z}_3^3, \mathbf{z}_1^3)\big)$.

4.6 Dot products

Due to the bilinearity of the dot-product operations one can simply replace the calls of ring tensor multiplication with calls to ring tensor dot product and the protocol will be valid as long as $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \boldsymbol{\alpha}_3$ have matching dimensions. For more details, the reader can check the `DotOp` implementation in `moose`.

4.7 Sharing data

In Figure 4.7.1 we show how a party D can translate a private tensor $\mathbf{x}$ (known only to D) into a replicated tensor. Here we distinguish two cases: a) when D is amongst the parties on the replicated placement i.e. $D \in \{P_1, P_2, P_3\}$ and b) when D is an external party, not in the replicated set.

In the first case the parties use a similar mechanism as in the ZeroShare to produce a set of replicated shared seeds. Wlog., consider $D = P_i$, then P_i sends to P_{i-1} the shape of $\mathbf{x}$ in order for P_{i-1} to derive a correct sized random tensor using the shared seed. Then the inputting party P_i masks the private tensor with a random value $\mathbf{x} - \mathbf{x}_i^i$ and sends it to P_{i+1}.

Correctness can be seen from the fact that at the end of the protocol all $\mathbf{x}_1^j + \mathbf{x}_2^j + \mathbf{x}_3^{j+1} = \mathbf{x}$ for all $j \in [1, 3]$. For example, when $j = 1$ we have the following:

$$\mathbf{x}_1^1 + \mathbf{x}_2^1 + \mathbf{x}_3^2 = \mathbf{0} + \mathbf{x}_2^1 + (\mathbf{x} - \mathbf{x}_2^2) = \mathbf{x}$$

due to $\mathbf{x}_2^1 = \mathbf{x}_2^2$ since they were sampled from identical seeds. Security follows from [ABF+16] or [MR18].

In the second case D is an external party sharing their input. Two of the parties (P_1, P_2) produce the randomness used to mask $\mathbf{x}$ in $\mathbf{x}_3 \leftarrow \mathbf{x} - \mathbf{x}_1 - \mathbf{x}_2$ such that any two parties can reconstruct the secret $\langle \mathbf{x} \rangle$ afterwards. Correctness can be seen as in the first case since all $\mathbf{x}_1^j + \mathbf{x}_2^j + \mathbf{x}_3^{j+1} = \mathbf{x}$ for all $j \in [1, 3]$. For example, when $j = 1$ we have:

$$\mathbf{x}_1^1 + \mathbf{x}_2^1 + \mathbf{x}_3^2 = \mathbf{x}_1 + \mathbf{x}_2 + (\mathbf{x} - \mathbf{x}_1 - \mathbf{x}_2) = \mathbf{x}.$$

The security argument differs slightly then the first case since D sends $\mathbf{x} - \mathbf{x}_1 - \mathbf{x}_2$ to P_2 and P_3. Since $\mathbf{x}_1$ and $\mathbf{x}_2$ are joint randomness produced by P_1 and P_2 then any two parties (no less) can reconstruct the secret.

4.7.1 $\mathsf{ReplicatedShare}_{[D,P_1,P_2,P_3]}(R; \mathbf{k}, \mathbf{x})$

Let D be the player holding $\mathbf{x}$. Let $\mathsf{sync\_key}_0$ and $\mathsf{sync\_key}_1$ be distinct values.

1. Let $\big((k_1^1, k_2^1), (k_2^2, k_3^2), (k_3^3, k_1^3)\big) = \mathbf{k}$ be PRF keys generated during setup.
2. If $D \in \{P_1, P_2, P_3\}$ let i the index for which $D = P_i$:
   (a) On P_i:
      i. $\mathsf{shape}^i \leftarrow \mathsf{Shape}(\mathbf{x})$.
      ii. $\mathsf{seed}^i \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_0; k_i^i)$.
      iii. $\mathbf{x}_i^i \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}^i, \mathsf{seed}^i)$.
      iv. $\mathbf{x}_{i+1}^i \leftarrow \mathbf{x} - \mathbf{x}_i^i$.
      v. Send shape^i to P_{i-1} and P_{i+1}.
      vi. Send $\mathbf{x}_{i+1}^i$ to P_{i+1}.
   (b) On P_{i-1}:
      i. Receive shape^{i-1} from P_i.
      ii. $\mathsf{seed}^{i-1} \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_0; k_i^{i-1})$.
      iii. $\mathbf{x}_{i-1}^{i-1} \leftarrow \mathsf{Zeros}(R; \mathsf{shape}^{i-1})$.
      iv. $\mathbf{x}_i^{i-1} \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}^{i-1}, \mathsf{seed}^{i-1})$.
   (c) On P_{i+1}:
      i. Receive shape^{i+1} from P_i.
      ii. Receive $\mathbf{x}_{i+1}^i$ from P_i.
      iii. $\mathbf{x}_{i+1}^{i+1} \leftarrow \mathbf{x}_{i+1}^i$.

iv. $\mathbf{x}_{i+2}^{i+1} \leftarrow \mathsf{Zeros}(R; \mathsf{shape}^{i+1})$.

3. If $D \notin \{P_1, P_2, P_3\}$ then:

   (a) On D compute $\mathsf{shape}^D \leftarrow \mathsf{Shape}(\mathbf{x})$ and broadcast shape^D to all $P_{i\in[3]}$.

   (b) On P_1:

      i. $\mathsf{seed}_0^1 \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_0, k_1^1)$.
      ii. $\mathsf{seed}_1^1 \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_1, k_2^1)$.
      iii. $\mathbf{x}_1^1 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}^D, \mathsf{seed}_0^1)$.
      iv. $\mathbf{x}_2^1 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}^D, \mathsf{seed}_1^1)$.
      v. Send seed_0^1 to D.

   (c) On P_2:

      i. $\mathsf{seed}_1^2 \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_1, k_2^2)$.
      ii. $\mathbf{x}_2^2 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}^D, \mathsf{seed}_2^2)$.
      iii. Send seed_2^2 to D

   (d) On D:

      i. Receive seed_0^1 from P_1 and seed_1^2 from P_2.
      ii. $\mathbf{x}_1 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}^D, \mathsf{seed}_0^1)$.
      iii. $\mathbf{x}_2 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}^D, \mathsf{seed}_1^2)$.
      iv. $\mathbf{x}_3 \leftarrow \mathbf{x} - \mathbf{x}_1 - \mathbf{x}_2$.
      v. Send $\mathbf{x}_3$ to P_2 and P_3.

   (e) On P_2:

      i. Receive $\mathbf{x}_3$ from D and set $\mathbf{x}_3^2 \leftarrow \mathbf{x}_3$.

   (f) On P_3.

      i. Receive $\mathbf{x}_3$ from D. Set $\mathbf{x}_3^3 \leftarrow \mathbf{x}_3$.
      ii. $\mathsf{seed}_0^3 \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_0, k_1^3)$.
      iii. $\mathbf{x}_1^3 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}^D, \mathsf{seed}_0^3)$.

4. Return $\langle \mathbf{x} \rangle_R = \big((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\big)$.

4.8 Revealing secrets

We sometimes use interchangeably the terms *open* and *reveal* a secret to refer to taking a replicated tensor $\langle \mathbf{x} \rangle$ and making its $\mathbf{x}$ known to a single party (Figure 4.8.1) or to all parties (Figure 4.8.2). Note that Figure 4.8.1 corresponds to a replicated opening, whereas Figure 4.8.2 corresponds to a two out of two additive sharing scheme. Additive sharing is used in the truncation and share conversion protocols below. Correctness is guaranteed as long as the secret was correctly shared. This is also secure, as we only send the required shares to the set of parties that want the secret revealed.

Note that we also have implemented a variant of revealing to an external party - or a host placement which is not on the replicated placement. This is a simple case where any two parties P_i, P_{i+1} send their shares $(x_i^i, x_{i+1}^i, x_{i+2}^{i+1})$ to the external party onto which are then added.

Replicated opening protocol to a specific party.

4.8.1 $\mathsf{Open}_{[P_1,P_2,P_3]}\left(\langle \mathbf{x} \rangle^{\mathsf{rep}}\right)$

Let P_i be the player to which $\mathbf{x}$ is to be revealed.

1. Let $\left((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\right) = \langle \mathbf{x} \rangle^{\mathsf{rep}}$.
2. On P_{i+1}:
   (a) Send $\mathbf{x}_{i+2}^{i+1}$ to P_i.
3. On P_i:
   (a) Receive $\mathbf{x}_{i+2}^{i+1}$ from P_i.
   (b) $\mathbf{x} \leftarrow \mathbf{x}_{i+2}^{i+1} + \mathbf{x}_i^i + \mathbf{x}_{i+1}^i$.
4. Return $\mathbf{x}$.

Additive opening protocol.

4.8.2 $\mathsf{Open}_{[P_1,P_2]}\left(\langle \mathbf{x} \rangle^{\mathsf{adt}}\right)$

1. Let $(\mathbf{x}_1, \mathbf{x}_2) = \langle \mathbf{x} \rangle^{\mathsf{adt}}$.
2. On P_i for $i \in [2]$:
   (a) Send $\mathbf{x}_i$ to P_{i+1}.
   (b) Receive $\mathbf{x}_{i+1}$ from P_{i+1}.
   (c) $\mathbf{x}^i \leftarrow \mathbf{x}_1 + \mathbf{x}_2$.
3. Return $\mathbf{x}^1$ and $\mathbf{x}^2$.

4.9 Truncation

Given a replicated tensor $\langle \mathbf{x} \rangle$, the goal of the truncation protocol is to compute $\langle \mathbf{x} \bmod 2^m \rangle$ where m is a public constant. For the semi-honest 3PC case we implement the probabilistic truncation from [DEK20b] where the preprocessing is done by one party (P_3 for eg) which is then secret shared to the other two parties (P_1 and P_2).

The protocol is described in Figure 4.9.1. The high-level idea is to convert the replicated sharing $\langle \mathbf{x} \rangle$ to a 2-out-of-2 additive sharing $\langle \mathbf{x} \rangle^{\mathsf{adt}}$ and then execute the truncation protocol between two parties (Figure 4.9.2). The preprocessing $\mathbf{r}, \mathbf{r}_{\mathsf{msb}}, \mathbf{r}_{\mathsf{top}}$ for the 2PC computation is generated by P_3 which is later additively shared to P_1 and P_2 (Figure 4.9.3). After the two-party truncation is done, the output share $\langle \mathbf{y}' \rangle^{\mathsf{adt}}$ is then converted back to a replicated sharing $\langle \mathbf{y} \rangle^{\mathsf{rep}}$ using $\mathsf{AdtToRep}$. Correctness and security follow from [DEK20b].

In our implementation of the additive to replicated share conversions we consider a more general variant, i.e. when an additive shared tensor $\langle \cdot \rangle^{\mathsf{adt}}$ is converted to a replicated secret $\langle \cdot \rangle^{\mathsf{rep}}$ where the host placements forming adt are not necessarily a subset of the host placements that define rep (see Protocol A.1.1).

4.9.1 $\mathsf{TruncPr}_{[P_1,P_2,P_3]}\left(\mathbb{Z}_{2^k}; \langle \mathbf{x} \rangle_{2^k}^{\mathsf{rep}}, m\right)$

Three party truncation protocol.

1. Let $\left((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\right) = \langle \mathbf{x} \rangle_{2^k}^{\mathsf{rep}}$.
2. On P_3:
   (a) $\mathsf{shape} \leftarrow \mathsf{Shape}(\mathbf{x}_3^3)$.

(b) $\mathbf{r} \leftarrow \mathsf{SampleUniform}(\mathbb{Z}_{2^k}; \mathsf{shape})$.

(c) $\mathbf{r}_{\mathsf{msb}} \leftarrow \mathsf{Shr}(k-1; \mathbf{r})$.

(d) $\mathbf{r}_{\mathsf{top}} \leftarrow \mathsf{Shr}(m+1; \mathsf{Shl}(1; \mathbf{r}))$.

3. $\langle \mathbf{r} \rangle^{\mathsf{adt}} \leftarrow \mathsf{AdditiveShare}_{[D=P_3,P_1,P_2]}(\mathbf{r})$
4. $\langle \mathbf{r}_{\mathsf{msb}} \rangle^{\mathsf{adt}} \leftarrow \mathsf{AdditiveShare}_{[D=P_3,P_1,P_2]}(\mathbf{r}_{\mathsf{msb}})$
5. $\langle \mathbf{r}_{\mathsf{top}} \rangle^{\mathsf{adt}} \leftarrow \mathsf{AdditiveShare}_{[D=P_3,P_1,P_2]}(\mathbf{r}_{\mathsf{top}})$
6. $\langle \mathbf{x}' \rangle^{\mathsf{adt}} \leftarrow \mathsf{ReplicatedToAdditive}_{[P_1,P_2]}(\langle \mathbf{x} \rangle^{\mathsf{rep}})$.
7. $\langle \mathbf{y}' \rangle^{\mathsf{adt}} \leftarrow \mathsf{2PCTruncPr}_{[P_1,P_2]}(\langle \mathbf{x}' \rangle^{\mathsf{adt}}, m, \langle \mathbf{r} \rangle^{\mathsf{adt}}, \langle \mathbf{r}_{\mathsf{msb}} \rangle^{\mathsf{adt}}, \langle \mathbf{r}_{\mathsf{top}} \rangle^{\mathsf{adt}})$.
8. Return $\langle \mathbf{y} \rangle^{\mathsf{rep}} \leftarrow \mathsf{AdtToRep}_{[P_1,P_2,P_3]}(\langle \mathbf{y}' \rangle^{\mathsf{adt}})$.

4.9.2 $\mathsf{2PCTruncPr}_{[P_1,P_2]}(\langle \mathbf{x} \rangle, m, \langle \mathbf{r} \rangle, \langle \mathbf{r}_{\mathsf{msb}} \rangle, \langle \mathbf{r}_{\mathsf{top}} \rangle)$

Two-party probabilistic truncation protocol. Assume $0 \leq \mathbf{x} < 2^{\log_2(R-1)}$ in the unsigned representation.

Since we deal with signed numbers the bounds are $-2^{\log_2(R-2)} \leq x < 2^{\log_2(R-2)}$ in order to have enough room to make $\mathbf{x}$ positive (i.e. add $2^{\log_2(R-2)}$) but also keep its MSB equal to 0.

Denote $k = \log_2(R) - 1$, considered in some literature as the unsigned input bound (in bits).

1. $\langle \mathbf{c} \rangle \leftarrow \langle \mathbf{x} \rangle + 2^{k-1} + \langle \mathbf{r} \rangle$. By adding 2^{k-1} we make sure we mask a positive number.
2. $\mathbf{c} \leftarrow \mathsf{Open}_{[P_1,P_2]}(\langle \mathbf{c} \rangle)$.
3. $\mathbf{c}_{\mathsf{top}} \leftarrow (\mathbf{c}/2^m) \bmod 2^{k-m-1}$.
4. $\mathbf{c}_{\mathsf{msb}} \leftarrow \mathbf{c}/2^k$.
5. $\langle \mathbf{w} \rangle \leftarrow \langle \mathbf{r}_{\mathsf{msb}} \rangle \oplus \mathbf{c}_{\mathsf{msb}}$. Note here that $a \oplus b = a + b - 2a \cdot b$.
6. $\langle \mathbf{y}_+ \rangle \leftarrow \mathbf{c}_{\mathsf{top}} - \langle \mathbf{r}_{\mathsf{top}} \rangle + \langle \mathbf{w} \rangle \cdot 2^{k-m}$.
7. Return $\langle \mathbf{y} \rangle \leftarrow \langle \mathbf{y}_+ \rangle - 2^{k-m-1}$.

4.9.3 $\mathsf{AdditiveShare}_{[D,P_1,P_2]}(R; \mathbf{x})$

Additive sharing protocol. Let D be the party holding $\mathbf{x}$.

1. On D:
   (a) $\mathsf{shape} \leftarrow \mathsf{Shape}(\mathbf{x})$.
   (b) $\mathbf{x}^1 \leftarrow \mathsf{SampleUniform}(R; \mathsf{shape})$.
   (c) $\mathbf{x}^2 \leftarrow \mathbf{x} - \mathbf{x}^1$.
   (d) Send $\mathbf{x}^1$ to P_1.
   (e) Send $\mathbf{x}^2$ to P_2.
2. On P_1:
   (a) Receive $\mathbf{x}_1^1$ from D.
3. On P_2:
   (a) Receive $\mathbf{x}_2^2$ from D.
4. Return $\langle \mathbf{x} \rangle^{\mathsf{adt}} = (\mathbf{x}_1^1, \mathbf{x}_2^2)$.

4.9.4 $\mathsf{ReplicatedToAdditive}_{[P_1,P_2]}\left(\langle \mathbf{x} \rangle^{\mathsf{rep}}\right)$

Conversion from a replicated share to an additive share.

1. Let $\left((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\right) = \langle \mathbf{x} \rangle^{\mathsf{rep}}$.
2. On P_1:
   (a) $\mathbf{x}'_1 \leftarrow \mathbf{x}_1^1 + \mathbf{x}_2^1$.
3. On P_2:
   (a) $\mathbf{x}'_2 \leftarrow \mathbf{x}_3^2$.
4. Return $\langle \mathbf{x}' \rangle^{\mathsf{adt}} = (\mathbf{x}'_1, \mathbf{x}'_2)$.

4.9.5 $\mathsf{AdtToRep}_{[P_1,P_2,P_3]}(R; \langle \mathbf{x} \rangle_R^{\mathsf{adt}})$

Additive to replicated share conversion protocol. Let $\mathsf{sync\_key}_1$ and $\mathsf{sync\_key}_2$ be distinct fixed values.

1. Let $(\mathbf{x}_1, \mathbf{x}_2) = \langle \mathbf{x} \rangle_R^{\mathsf{adt}}$.
2. On P_1:
   (a) $\mathsf{shape} \leftarrow \mathsf{Shape}(\mathbf{x}_1)$.
   (b) Send shape to P_3.
3. On P_3:
   (a) Receive shape from P_1.
   (b) $k \leftarrow \mathsf{GenPrfKey}()$.
   (c) $\mathsf{seed}_i \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_i; k)$ for $i \in \{1, 2\}$.
   (d) Send seed_i to P_i for $i \in \{1, 2\}$.
   (e) $\mathbf{y}_1^3 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}, \mathsf{seed}_1)$.
   (f) $\mathbf{y}_3^3 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}, \mathsf{seed}_2)$.
4. On P_i for $i \in \{1, 2\}$:
   (a) $\mathsf{shape} \leftarrow \mathsf{Shape}(\mathbf{x}_i)$.
   (b) Receive seed_i from P_3.
   (c) $\mathbf{y}_i \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}, \mathsf{seed}_i)$.
5. Let $\langle \mathbf{y} \rangle_R^{\mathsf{adt}} = (\mathbf{y}_1, \mathbf{y}_2)$.
6. $\mathbf{c} \leftarrow \mathsf{Open}_{[P_1,P_2]}\left(\langle \mathbf{x} \rangle_R^{\mathsf{adt}} - \langle \mathbf{y} \rangle_R^{\mathsf{adt}}\right)$
7. Return $\langle \mathbf{y} \rangle_R^{\mathsf{rep}} = \left((\mathbf{y}_1, \mathbf{c}), (\mathbf{c}, \mathbf{y}_2), (\mathbf{y}_3^3, \mathbf{y}_1^3)\right)$.

4.10 Comparison

The problem of computing secure comparisons translates directly into computing a sharing of the most significant bit $\langle \mathsf{msb}(\mathbf{x}) \rangle$. In the 3PC semi-honest model there are few approaches to this:

1. ABY3 [MR18]. Each party locally bit-decomposes their shares over $\mathbb{Z}_{2^k}$ and then reconstruct the secret modulo $\mathbb{Z}_{2^k}$ using shares from $\mathbb{Z}_2^k$ and a binary adder. Once a boolean sharing of the MSB is computed using the binary adder this is converted to a ring sharing in $\mathbb{Z}_{2^k}$. The share conversion was originally done using a three-party OT protocol. Later, in MP-SPDZ [Kel20] the boolean to ring secret share conversion was achieved with a daBit.

2. SecureNN [WGC19]. Similar to ABY3 with the difference that it uses arithmetic modulo small fields but avoids ring-to-boolean conversion.

3. Extended daBits [EGK+20]. The 3PC case has roughly the same cost as ABY3.

In moose we use the MSB extraction protocol from ABY3 with a Kogge-Stone binary adder and minor optimizations for tensor operations. We avoid the special three-party OT that ABY3 as well as edaBits preprocessing by introducing our custom protocol B2A for binary to ring share conversion in Figure 4.10.4. The MSB protocol can be found in Figure 4.10.2 where $\bar{\cdot}$ is used to denote a vector with k elements indexed using $\bar{\cdot}[i]$ for $i \in [k]$.

4.10.1 $\mathsf{BitDecRaw}_{[P_1,P_2,P_3]}\left(\langle \mathbf{x} \rangle_{2^k}^{\mathsf{rep}}\right) \rightarrow [\langle \cdot \rangle_2^{\mathsf{rep}}; k]$

Ring bit decomposition to binary shares.

1. Let $\left((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\right) = \langle \mathbf{x} \rangle_{2^k}^{\mathsf{rep}}$.
2. On P_1:
   (a) $\bar{\mathbf{a}} \leftarrow \mathsf{LocalBitDec}(\mathbf{x}_1^1 + \mathbf{x}_2^1)$.
   (b) $\overline{\langle \mathbf{a} \rangle_2}[i] \leftarrow \mathsf{ReplicatedShare}(\mathbb{Z}_2; \bar{\mathbf{a}}[i])$ for $i \in [k]$.
3. On P_2:
   (a) $\overline{\mathbf{b}^2} \leftarrow \mathsf{LocalBitDec}(\mathbf{x}_3^2)$.
4. On P_3:
   (a) $\overline{\mathbf{b}^3} \leftarrow \mathsf{LocalBitDec}(\mathbf{x}_3^3)$.
5. Let $\overline{\langle \mathbf{b} \rangle_2}[i] = \left((0,0), (0, \overline{\mathbf{b}^2}[i]), (\overline{\mathbf{b}^3}[i], 0)\right)$ for $i \in [k]$.
6. Return $\mathsf{BinaryAdder}(\overline{\langle \mathbf{a} \rangle_2}, \overline{\langle \mathbf{b} \rangle_2})$.

4.10.2 $\mathsf{MSB}_{[P_1,P_2,P_3]}\left(\langle \mathbf{x} \rangle_{2^k}^{\mathsf{rep}}\right)$

Most significant bit from a replicated ring sharing.

1. $\overline{\langle \mathbf{b} \rangle} \leftarrow \mathsf{BitDecRaw}(\langle \mathbf{x} \rangle)$.
2. Return $\mathsf{B2A}(\overline{\langle \mathbf{b} \rangle_2}[k-1])$.

4.10.3 New boolean to ring sharing protocol

When the inputs are tensors we can perform the share conversion $\langle\cdot\rangle_2 \mapsto \langle\cdot\rangle_{2^k}$ (B2A function) more efficient by making use of the fact that all parties follow the protocol specifications. The starting point is using a similar idea from daBit/edaBit [RW19, EGK+20] line of work with the twist that P_3 generates the preprocessing material.

The protocol works as follows: P_3 acts as a trusted third party and generates a random daBit $(\langle\mathbf{b}\rangle_2, \langle\mathbf{b}\rangle_{2^k})$ locally and shares it to P_1 and P_2. Note that P_3 uses the seed derivation to to minimize the communication between P_3 and the rest of the parties when sharing the random daBit.

Next, P_1 and P_2 run a two-party protocol to convert $\langle\mathbf{x}\rangle_2$ to $\langle\mathbf{x}\rangle_{2^k}$ by computing the boolean XOR $\langle\mathbf{c}\rangle_2 \leftarrow \langle\mathbf{x}\rangle_2 \oplus \langle\mathbf{b}\rangle_2$. Then the two parties P_1 and P_2 reveal $\langle\mathbf{c}\rangle_2$ to the random daBit with an XOR in the arithmetic domain $\mathbf{x}_{2^k} = \mathbf{c} + \mathbf{b}_{2^k} - 2 \cdot \mathbf{c} \cdot \mathbf{b}_{2^k}$. This leaks no information $\mathbf{b}$ mask uniformly random the input $\mathbf{x}$. Finally they securely convert back the additive sharing of $\mathbf{x}_{2^k}$ denoted as $\langle\mathbf{x}\rangle_{2^k}^{\mathsf{adt}}$ to a replicated sharing $\langle\mathbf{x}\rangle_{2^k}^{\mathsf{rep}}$ using AdtToRep protocol. We fully describe this share conversion protocol in Figure 4.10.4.

4.10.4 $\mathsf{B2A}_{[P_1,P_2,P_3]}\left(\langle\mathbf{x}\rangle_2^{\mathsf{rep}}\right)$

Binary to arithmetic share conversion.

1. Let $\left((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\right) = \langle\mathbf{x}\rangle_2^{\mathsf{rep}}$.
2. On P_1:
   (a) $\mathbf{x}_{12}^1 \leftarrow \mathbf{x}_1^1 \oplus \mathbf{x}_2^1$.
   (b) $\mathsf{shape} \leftarrow \mathsf{Shape}(\mathbf{x}_1^1)$.
3. Let $\langle\mathbf{x}\rangle_2^{\mathsf{adt}} = (\mathbf{x}_{12}^1, \mathbf{x}_3^2)$.
4. $\langle\mathbf{b}\rangle_2^{\mathsf{adt}}, \langle\mathbf{b}\rangle_{2^k}^{\mathsf{adt}} \leftarrow \mathsf{DaBit}_{[D=P_3,P_1,P_2]}(\mathsf{shape})$.
5. $\langle\mathbf{c}\rangle_2^{\mathsf{adt}} \leftarrow \langle\mathbf{b}\rangle_2^{\mathsf{adt}} \oplus \langle\mathbf{x}\rangle_2^{\mathsf{adt}}$.
6. $\mathbf{c} \leftarrow \mathsf{Open}_{[P_1,P_2]}(\langle\mathbf{c}\rangle_2^{\mathsf{adt}})$.
7. $\langle\mathbf{x}\rangle_{2^k}^{\mathsf{adt}} \leftarrow \mathbf{c} + \langle\mathbf{b}\rangle_{2^k}^{\mathsf{adt}} - 2 \cdot \mathbf{c} \cdot \langle\mathbf{b}\rangle_{2^k}^{\mathsf{adt}}$.
8. Return $\mathsf{AdtToRep}_{[P_1,P_2,P_3]}(\langle\mathbf{x}\rangle_{2^k}^{\mathsf{adt}})$.

4.10.5 $\mathsf{DaBit}_{[D,P_1,P_2]}(\mathsf{shape})$

Let $\mathsf{sync\_key}_0, \mathsf{sync\_key}_1, \mathsf{sync\_key}_2$ be distinct fixed values.

1. On D:
   (a) $\mathsf{key} \xleftarrow{\$} \mathsf{GenPrfKey}()$.
   (b) $\mathsf{seed} \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_0; \mathsf{key})$.
   (c) $\mathsf{seed}_2 \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_1, \mathsf{key})$.
   (d) $\mathsf{seed}_{2^k} \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_2, \mathsf{key})$.
   (e) Send seed_2 and seed_{2^k} to P_1.
   (f) $\mathbf{b} \leftarrow \mathsf{SampleUniformSeeded}(\mathbb{Z}_2; \mathsf{seed}, \mathsf{shape})$.
   (g) $\mathbf{b}_2^1 \leftarrow \mathsf{SampleUniformSeeded}(\mathbb{Z}_2; \mathsf{seed}_2, \mathsf{shape})$.
   (h) $\mathbf{b}_2^2 \leftarrow \mathbf{b} - \mathbf{b}_2^1$.

(i) $\mathbf{b}^1_{2^k} \leftarrow \mathsf{SampleUniformSeeded}(\mathbb{Z}_{2^k}; \mathsf{seed}_{2^k}, \mathsf{shape})$.

(j) $\mathbf{b}^2_{2^k} \leftarrow \mathbf{b} - \mathbf{b}^1_{2^k}$.

(k) Send $\mathbf{b}^2_2$ and $\mathbf{b}^2_{2^k}$ to P_2.

2. On P_1:

   (a) Receive seed_2 and seed_{2^k} from D.

   (b) $\mathbf{b}^1_2 \leftarrow \mathsf{SampleUniformSeeded}(\mathbb{Z}_2; \mathsf{seed}_2, \mathsf{shape})$.

   (c) $\mathbf{b}^1_{2^k} \leftarrow \mathsf{SampleUniformSeeded}(\mathbb{Z}_{2^k}; \mathsf{seed}_{2^k}, \mathsf{shape})$.

3. On P_2:

   (a) Receive $\mathbf{b}^2_2$ and $\mathbf{b}^2_{2^k}$ from D.

4. Let $\langle \mathbf{b} \rangle^{\mathsf{adt}}_2 = (\mathbf{b}^1_2, \mathbf{b}^2_2)$.

5. Let $\langle \mathbf{b} \rangle^{\mathsf{adt}}_{2^k} = (\mathbf{b}^1_{2^k}, \mathbf{b}^2_{2^k})$

6. Return $\langle \mathbf{b} \rangle^{\mathsf{adt}}_2$ and $\langle \mathbf{b} \rangle^{\mathsf{adt}}_{2^k}$.

4.10.6 $\mathsf{Abs}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle_{2^k})$

Absolute value computation from a replicated sharing.

1. $\langle \mathbf{b} \rangle_{2^k} \leftarrow \mathsf{MSB}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle_{2^k})$.
2. $\langle \mathbf{s} \rangle_{2^k} \leftarrow 1 - 2 \cdot \langle \mathbf{b} \rangle_{2^k}$.
3. Return $\langle \mathbf{s} \rangle_{2^k} \cdot \langle \mathbf{x} \rangle_{2^k}$.

4.11 Fixed-point division

The division algorithm was firstly introduced by Catrina and Saxena [CS10]. A few modifications to it were brought by MP-SPDZ to work over ring based circuits. We present the division algorithm in Figure 4.11.1 below.

4.11.1 $\mathsf{Div}_{[P_1,P_2,P_3]}(\langle \mathbf{a} \rangle_{k,f}, \langle \mathbf{b} \rangle_{k,f})$

1. Compute $\theta = \log_2(k/3.5)$.
2. Set $\boldsymbol{\alpha} = \mathsf{Fill}(2^{2\cdot f}, \mathsf{shape}(\langle \mathbf{a} \rangle))$. Note that $\boldsymbol{\alpha} = 1 \in \mathbb{Q}_{\langle 2\cdot k, 2\cdot f \rangle}$ representation.
3. $\langle \mathbf{w} \rangle = \mathsf{AppRcr}(\langle \mathbf{b} \rangle, k, f)$.
4. $\langle \mathbf{x} \rangle = \boldsymbol{\alpha} - \langle \mathbf{b} \rangle \cdot \langle \mathbf{w} \rangle$.
5. $\langle \mathbf{y} \rangle = \langle \mathbf{a} \rangle \cdot \langle \mathbf{w} \rangle$.
6. $\langle \mathbf{y} \rangle = \mathsf{Round}(\langle \mathbf{y} \rangle, 2 \cdot k, f)$.
7. For $i \in [0, \theta - 1]$:
   (a) $\langle \mathbf{y} \rangle = \langle \mathbf{y} \rangle \cdot (\boldsymbol{\alpha} + \langle \mathbf{x} \rangle)$.
   (b) $\langle \mathbf{x} \rangle = \langle \mathbf{x} \rangle \cdot \langle \mathbf{x} \rangle$.
   (c) $\langle \mathbf{y} \rangle = \mathsf{Round}(\langle \mathbf{y} \rangle, 2 \cdot k, 2 \cdot f)$.
   (d) $\langle \mathbf{x} \rangle = \mathsf{Round}(\langle \mathbf{x} \rangle, 2 \cdot k, 2 \cdot f)$.
8. $\langle \mathbf{y} \rangle = \langle \mathbf{y} \rangle \cdot (\boldsymbol{\alpha} + \langle \mathbf{x} \rangle)$.
9. $\langle \mathbf{y} \rangle = \mathsf{Round}(k + 2 \cdot f, 2 \cdot f)$.
10. Return $\langle \mathbf{y} \rangle_{k,f} = \langle \mathbf{y} \rangle$.

4.11.2 $\mathsf{AppRcr}_{[P_1,P_2,P_3]}(R; \langle \mathbf{b} \rangle^{\mathsf{rep}}_{2^k}, k, f)$

Outputs a secret $\langle \mathbf{w} \rangle$ such that $\mathbf{w} \approx 1/\mathbf{b}$, i.e. $\mathbf{w} \cdot \mathbf{b} \approx 1$.

1. Compute $\boldsymbol{\alpha} = \mathsf{Fill}(2.9142 \cdot 2^k, \mathsf{shape}(\langle \mathbf{b} \rangle))$. Here $\boldsymbol{\alpha} = 2.9142 \in \mathbb{Q}_{\langle k+f,k \rangle}$.
2. $\langle \mathbf{c} \rangle, \langle \mathbf{v} \rangle = \mathsf{Norm}(\langle \mathbf{b} \rangle, k, f)$.
3. $\langle \mathbf{d} \rangle = \boldsymbol{\alpha} - 2 \cdot \langle \mathbf{c} \rangle$.
4. $\langle \mathbf{w} \rangle = \langle \mathbf{d} \rangle \cdot \langle \mathbf{v} \rangle$.
5. $\langle \mathbf{w} \rangle = \mathsf{Round}(\langle \mathbf{w} \rangle, 2 \cdot k, 2 \cdot (k - f))$. (get rid of the integral part since we want an output between $[0, 1]$)
6. Return $\langle \mathbf{w} \rangle$.

4.11.3 $\mathsf{Norm}_{[P_1,P_2,P_3]}(R; \langle \mathbf{b} \rangle^{\mathsf{rep}}_{2^k}, k, f)$

Outputs a secret shared tuple $(\langle \mathbf{c} \rangle, \langle \mathbf{v} \rangle)$ such that $2^{k-1} \leq \mathbf{c} < 2^k$ and $\mathbf{c} = \mathbf{b} \cdot \mathbf{v}$.

1. $\langle \mathsf{sgn} \rangle = 1 - 2 \cdot \mathsf{LTZ}(\langle \mathbf{b} \rangle)$.
2. $\langle \mathbf{x} \rangle = \langle \mathsf{sgn} \rangle \cdot \langle \mathbf{b} \rangle$.
3. $\langle \mathbf{x}_0 \rangle, \ldots, \langle \mathbf{x}_{k-1} \rangle = \mathsf{BitDec}(\mathbf{x}, k)$.
4. $\langle \mathbf{y}_0 \rangle, \ldots, \langle \mathbf{y}_{k-1} \rangle = \mathsf{Rev}(\mathsf{PreOR}(\langle \mathbf{x}_{k-1} \rangle, \ldots, \langle \mathbf{x}_0 \rangle))$.
5. For $i \in [0, k-2]$:
   (a) $\langle \mathbf{z}_i \rangle = \langle \mathbf{y}_i \rangle - \langle \mathbf{y}_{i+1} \rangle$
6. $\langle \mathbf{z}_{k-1} \rangle = \langle \mathbf{y}_{k-1} \rangle$.
7. $\langle \mathbf{s} \rangle \leftarrow \sum_{i=0}^{k-1} 2^{k-i-1} \cdot \langle \mathbf{z}_i \rangle$.
8. $\langle \mathbf{c} \rangle = \langle \mathbf{x} \rangle \cdot \langle \mathbf{s} \rangle$.
9. $\langle \mathbf{v} \rangle = \langle \mathsf{sgn} \rangle \cdot \langle \mathbf{s} \rangle$.
10. Return $(\langle \mathbf{c} \rangle, \langle \mathbf{v} \rangle)$.

4.11.4 $\mathsf{BitDec}_{[P_1,P_2,P_3]}(R; \langle \mathbf{x} \rangle^{\mathsf{rep}}_{2^k}, k)$

1. $(\langle \mathbf{b}_0 \rangle_2, \ldots, \langle \mathbf{b}_{k-1} \rangle_2) \leftarrow \mathsf{BitDecRaw}(\mathbf{x}^{\mathsf{rep}}_{2^k})$.
2. Denote $\overline{\langle \mathbf{b} \rangle_2} = [\langle \mathbf{b}_0 \rangle_2, \ldots, \langle \mathbf{b}_{k-1} \rangle_2]$ as the tensor containing $\mathbb{Z}_2$ shares of $\mathbf{x}$'s bits.
3. Call $\overline{\langle \mathbf{c} \rangle} = \mathsf{B2A}(\overline{\langle \mathbf{b} \rangle_2})$ where $\langle \mathbf{c} \rangle : \langle \cdot \rangle_{\mathbb{Z}_{2^k}}$.
4. Return $(\overline{\langle \mathbf{c} \rangle(0)}, \ldots, \overline{\langle \mathbf{c} \rangle(k-1)})$.

4.11.5 $\mathsf{Round}_{[P_1,P_2,P_3]}(R; \langle \mathbf{x} \rangle^{\mathsf{rep}}_{R}, k, m)$

Rounding a k bit integer by m bits.

1. If numbers are unsigned assert $k + 1 \leq \log_2 R$. Otherwise assert $k + 1 < \log_2 R$.
2. Output $\mathsf{TruncPr}(\langle \mathbf{x} \rangle^{\mathsf{rep}}, m)$.

4.11.6 $\mathsf{PreOr}_{[P_1,P_2,P_3]}(R; \langle \mathbf{x} \rangle^{\mathsf{rep}}_{R})$

PrefixOR operation.

1. $n = \langle \mathbf{x} \rangle.\mathsf{len}$.
2. $\ell = \lceil \log_2 n \rceil$.
3. $\langle \mathbf{t} \rangle = \langle \mathbf{x} \rangle$.
4. For $i \in [0, \ell - 1]$:
   (a) For $j \in [0, 2^\ell / 2^{i+1}]$:
      i. $y = 2^i + j \cdot 2^{i+1} - 1$.
      ii. For $k \in [1, 2^i]$:
         A. Note that $\mathbf{a} | \mathbf{b} = \mathbf{a} + \mathbf{b} - \mathbf{a} \cdot \mathbf{b}$.
         B. If $y + k < n$ then $\langle \mathbf{t}_{y+k} \rangle = \langle \mathbf{t}_y \rangle | \langle \mathbf{t}_{y+k} \rangle$.
5. Return $\langle \mathbf{t} \rangle$.

4.12 Exponentiation

These methods are inspired from Aly and Smart [AS19] and few optimizations from MP-SPDZ.

4.12.1 $\mathsf{Exp}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle_{k,f})$

Computes $e^{\mathbf{x}} = 2^{\mathbf{x} \cdot \log_2 e}$.

1. Compute $\mathbf{c} = \log_2 e$.
2. $\langle \mathbf{s} \rangle_{k,f} = \mathbf{c}_{k,f} \cdot \langle \mathbf{x} \rangle_{k,f}$.
3. Return $\mathsf{Pow2}(\langle \mathbf{s} \rangle_{k,f})$.

4.12.2 $\mathsf{Pow2}_{[P_1,P_2,P_3]}(R; \langle \mathbf{a} \rangle_{k,f})$

Computes $\langle 2^{\mathbf{a}} \rangle_{k,f}$.

1. $n_{\mathsf{int}} = \lceil \log_2(k-f) \rceil$.
2. $n_{\mathsf{bits}} = n_{\mathsf{int}} + f$.
3. $\langle \mathbf{b} \rangle_2 = \mathsf{BitDec}(\langle \mathbf{a} \rangle)$.
4. $\langle \mathsf{msb} \rangle_R = \mathsf{B2A}(\langle \mathbf{b} \rangle[-1])$.
5. $\langle \mathbf{a}_+ \rangle_R = \mathsf{Mux}(\langle \mathsf{msb} \rangle_R, \langle \mathbf{a} \rangle_R, \langle -\mathbf{a} \rangle_R)$.
6. $\langle \mathsf{higher} \rangle_R = \mathsf{B2A}(\langle \mathbf{b} \rangle[f : n_{\mathsf{bits}}])$.
7. $\langle \mathsf{higher}_{\mathsf{int}} \rangle_R = \sum_{i=0}^{n_{\mathsf{int}}} (\langle \mathsf{higher}[i] \rangle_R \cdot 2^{f+i})$.
8. $\langle \mathsf{fract} \rangle_R = \langle \mathbf{a}_+ \rangle_R - \langle \mathsf{higher}_{\mathsf{int}} \rangle_R$.
9. $\langle \mathbf{d} \rangle_R = \mathsf{Pow2FromBits}(\langle \mathsf{higher} \rangle_R)$.
10. $\langle \mathbf{g} \rangle_{k,f} = \mathsf{ExpFromParts}(\langle \mathbf{d} \rangle_R, \langle \mathsf{fract} \rangle_R)$.
11. $\langle \mathbf{g}^{-1} \rangle_{k,f} = 1_{k,f} / \langle \mathbf{g} \rangle_{k,f}$.
12. $\langle \mathbf{r} \rangle_R = \mathsf{Mux}(\langle \mathsf{msb} \rangle_R, \langle \mathbf{g}^{-1} \rangle_R, \langle \mathbf{g} \rangle_R)$.
13. Return $\langle \mathbf{r} \rangle_{k,f}$.

4.12.3 $\mathsf{Pow2FromBits}_{[P_1,P_2,P_3]}(R; [\langle \mathbf{b} \rangle_R; \ell])$

Computes $\langle 2^{\sum_i 2^i \cdot \mathbf{b}[i]} \rangle$.

1. Let $\langle \mathbf{t} \rangle_R$ an ℓ dimensional vector filled with shared ring R tensors.
2. For $i \in \ell$:
   (a) $\langle \mathbf{t} \rangle[i] = \langle \mathbf{b} \rangle[i] \ll i + (1 - \langle \mathbf{b} \rangle[i])$.
3. Return $\mathsf{Reduce}(\langle \mathbf{t} \rangle, \cdot)$.

4.12.4 $\mathsf{ExpFromParts}_{[P_1,P_2,P_3]}(R; \langle \mathbf{e}_{\mathsf{int}} \rangle_R; \langle \mathbf{e}_{\mathsf{frac}} \rangle_{k,f})$

Computes exponentiation by dividing the exponent into integral and fractional part.

1. Let $\mathsf{amount} = k - 2 - f$.
2. $\langle x \rangle_{k,k-2} = \langle \mathbf{e}_{\mathsf{frac}} \rangle_R \ll \mathsf{amount}$.
3. $\langle \mathbf{e} \rangle_{k,k-2} = \mathsf{PolyEval}(p_{1045}, \langle x \rangle)$.
4. $\langle \mathbf{e}_{\mathsf{final}} \rangle_R = \mathsf{TruncPr}(\langle \mathbf{e} \rangle_R \cdot \langle \mathbf{e}_{\mathsf{int}} \rangle_R, \mathsf{amount})$.
5. Return $\langle \mathbf{e}_{\mathsf{final}} \rangle_{k,f}$.

4.12.5 $\mathsf{PolyEval}(R; \mathsf{coeffs}, \langle \mathbf{x} \rangle_{k,f})$

Secret polynomial evaluation in point $\mathbf{x}$.

1. Let ℓ the length of coeffs.
2. $d = \mathrm{argmin}_i(\mathsf{coeffs}[j] < 2^{-f+1})\ \forall j \in [i, \ell]$ (ignore small coefficients)
3. $\langle \mathbf{m} \rangle_{k,f} = \mathsf{PrefixOp}([\langle \mathbf{x} \rangle, \ldots, \langle \mathbf{x} \rangle, \cdot])$ where $\langle \mathbf{x} \rangle_{k,f}$ is cloned d times.
4. For $i \in [1, d]$:
   (a) $\langle s \rangle_R \leftarrow \langle s \rangle_R + \mathsf{coeffs}[i] \cdot \langle \mathbf{m} \rangle_R[i]$.
5. $\langle \mathbf{r} \rangle_{k,f} = \mathsf{TruncPR}(\langle s \rangle_R, f)$.
6. Return $\langle \mathbf{r} \rangle_{k,f} + \mathsf{coeffs}[0]$.

4.13 Softmax

We use Aly and Smart subprotocols since various MPC-friendly softmax approximations are comparable with the more precise variant (see Keller and Sun [KS21]). As in MP-SPDZ we chose to implement the softmax function using the exponentiation protocol described above.

4.13.1 $\mathsf{Softmax}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle_{k,f})$

Computes $e^{\mathbf{x}} / \sum \mathbf{x}_i$.

1. $\langle e^{\mathbf{x}} \rangle_{k,f} = \mathsf{Exp}(\mathbf{x})$.
2. Return $\langle e^{\mathbf{x}} \rangle_{k,f} / \mathsf{sum}(\langle \mathbf{x} \rangle_{k,f}, \mathsf{axis} : -1)$

The normalized version of the softmax is computed:

4.13.2 $\mathsf{NormedSoftmax}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle_{k,f})$

Computes $e^{\mathbf{x}} / \sum \mathbf{x}_i$.

1. $\langle \mathbf{t} \rangle_{k,f} \leftarrow e^{\langle \mathbf{x} \rangle_{k,f} - \mathsf{max}(\langle \mathbf{x} \rangle_{k,f}, -1)}$.
2. Return $\langle \mathbf{t} \rangle_{k,f} / \mathsf{sum}(\langle \mathbf{x} \rangle_{k,f}, -1)$.

In the example above the max function has an extra parameter, called axis, for more details the reader can check the numpy max function. At the lowest level, the max function between two values is implemented as the following:

4.13.3 $\mathsf{Maximum}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle_{2^k}, \langle \mathbf{y} \rangle_{2^k})$

1. Let $\langle \mathbf{b} \rangle_2 \leftarrow \langle \mathbf{x} \rangle_{2^k} < \langle \mathbf{y} \rangle_{2^k}$.
2. $\langle \mathbf{b} \rangle_{2^k} \leftarrow \langle \mathbf{b} \rangle_2$.
3. $\langle \mathbf{b} \rangle_{2^k} \cdot \langle \mathbf{y} \rangle_{2^k} + (1 - \langle \mathbf{b} \rangle_{2^k}) \cdot \langle \mathbf{x} \rangle_{2^k}$.
4. Return $\langle \mathbf{b} \rangle_{2^k}$.

4.14 Logarithm

We follow the same blueprint as Aly and Smart protocol for logarithm. In order to compute $\log_b \langle \mathbf{x} \rangle$ where $\mathbf{x}$ is secret, we first compute $\mathbf{c} = \log_2 \langle \mathbf{x} \rangle$ and then output $\mathbf{c} \cdot \log_2 b$. Next, the procedure $\mathsf{Int2FL}$ is used to normalize $\log_2 \langle \mathbf{x} \rangle$ into $[0.5, 1]$ which is then forwarded to a Pade polynomial approximation using polynomials p_{2524} and q_{2524}.

4.14.1 $\mathsf{Log}_{[P_1,P_2,P_3]}(\mathbf{b}, \langle \mathbf{x} \rangle_{k,f})$

Computes $\log_{\mathbf{b}} \mathbf{x}$.

1. $\mathbf{c} \leftarrow \log_2 \mathbf{b}$.
2. Return $\mathbf{c}_{k,f} \cdot \mathsf{Log2}(\langle \mathbf{x} \rangle_{k,f})$.

4.14.2 $\mathsf{Log2}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle_{k,f})$

Computes $\log_2 \mathbf{x}$.

1. $(\langle \mathbf{v} \rangle_R, \langle \mathbf{p} \rangle_R, \langle \mathbf{s} \rangle_R, \langle \mathbf{z} \rangle_R) \leftarrow \mathsf{Int2FL}(\langle \mathbf{x} \rangle_R, k, f)$.
2. $\langle \mathbf{p}_{2524} \rangle_{k,f} \leftarrow \mathsf{PolyEval}(p_{2524}, \langle \mathbf{v} \rangle_{k,f})$.
3. $\langle \mathbf{q}_{2524} \rangle_{k,f} \leftarrow \mathsf{PolyEval}(q_{2524}, \langle \mathbf{v} \rangle_{k,f})$.
4. Return $\langle \mathbf{p} \rangle_{k,f} + \langle \mathbf{p}_{2524} \rangle_{k,f} / \langle \mathbf{q}_{2524} \rangle_{k,f}$.

4.14.3 $\mathsf{Int2FL}_{[P_1,P_2,P_3]}(\langle \mathbf{x} \rangle_R, k, f)$

Computes $\mathbf{v}, \mathbf{p}, \mathbf{s}, \mathbf{z}$ such that $(1 - 2\mathbf{s}) \cdot (1 - \mathbf{z}) \cdot \mathbf{v} \cdot 2^{\mathbf{p}} = \mathbf{x}$.

1. $\langle \mathbf{s} \rangle_{2^k} = \langle \mathbf{a} \rangle_{2^k} < 0$.
2. $\langle \mathbf{z} \rangle_{2^k} = \mathsf{Equal}(\langle \mathbf{a} \rangle_{2^k}, 0)$.
3. $\langle \mathbf{a}^+ \rangle_{2^k} = \mathsf{Mux}(\langle \mathbf{s} \rangle_{2^k}, \langle -\mathbf{a} \rangle_{2^k}, \langle \mathbf{a} \rangle_{2^k})$.
4. $\langle \mathbf{a}_0^+ \rangle_2, \ldots, \langle \mathbf{a}_{k-1}^+ \rangle_2 = \mathsf{BitDec}(\langle \mathbf{a}^+ \rangle_{2^k}, k-1)$.
5. $\langle \mathbf{b}_0 \rangle_2, \ldots, \langle \mathbf{b}_{k-1} \rangle_2 = \mathsf{PreOr}(\langle \mathbf{a}_{k-1}^+ \rangle_2, \ldots, \langle \mathbf{a}_0^+ \rangle_2)$.
6. $\langle \mathbf{t} \rangle_{2^k} = \langle \mathbf{a}^+ \rangle_{2^k} \cdot (1 + \sum_{i=0}^{k-1} (2^i \cdot (1 - \langle \mathbf{b}_i \rangle_{2^k}))$.
7. $\langle \mathbf{p} \rangle_{2^k} = \sum_{i=0}^{k-1} \langle \mathbf{b}_i \rangle_{2^k}$.
8. $\langle \mathbf{v} \rangle_{2^k} = \mathsf{Round}(\langle \mathbf{t} \rangle_{2^k}, k, f)$.
9. $\langle \mathbf{p} \rangle_{2^k} = (\langle \mathbf{p} \rangle_{2^k} - f) \cdot (1 - \langle \mathbf{z} \rangle_{2^k})$.
10. Return $\langle \mathbf{v} \rangle_{2^k}, \langle \mathbf{p} \rangle_{2^k}, \langle \mathbf{s} \rangle_{2^k}, \langle \mathbf{z} \rangle_{2^k}$.

5 Implementation

5.1 Runtime

Operations on ring tensors are implemented using the `ndarray` Rust crate on `Wrapping<u64>` and `Wrapping<u128>` scalars. Computations are executed asynchronously via the `tokio` Rust crate by spawning a task for each operations.

5.2 AES based PRG

To efficiently sample a large batch of random bits, we use an AES-based PRG. We can produce $n \cdot 128$ random bits by calling $\mathsf{AES}_k(0), \cdots, \mathsf{AES}_k(n-1)$ with a 128 bit key k. Since AES works on 128 bit blocks, the integer counters $0, \cdots, (n-1)$ are encoded using Little Endian format. In case one wants to retrieve a number of ℓ random bits which is non-divisible by 128, the last $128 - \ell$ bits are discarded.

This method of generating randomness is used in other well-known MPC libraries, such as MP-SPDZ [Kel20], SCALE-MAMBA [ACC+21] or Swanky [Gal21]. Security proofs for using fixed key AES in various MPC protocols were given in [GKWY20].

The PRNG wrapper we have built can be found in `rust/src/prng.rs` and it is based on [Art21]. We set the `AES` crate to compile with special SSE3 CPU instructions whenever possible. The PRNG key k is generated using a call to `randombytes_into` from the Sodium Oxide rust crate [dna21].

References

[ABC+16] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[ABF+16] Toshinori Araki, Assaf Barak, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. DEMO: High-throughput secure three-party computation of kerberos ticket generation. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1841–1843. ACM Press, October 2016.

[ACC+21] Abdelrahaman Aly, Kelong Cong, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P. Smart, Titouan Tanguy, and Tim Wood. SCALE-MAMBA v1.12: Documentation, 2021.

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

[Art21] Artyom Pavlov. Crate aes. https://docs.rs/aes/0.6.0/aes/index.html, 2021. Online; accessed 2021.

[AS19] Abdelrahaman Aly and Nigel P. Smart. Benchmarking privacy preserving scientific operations. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 509–529. Springer, Heidelberg, June 2019.

[Bla22] Blake3 team. Blake3. https://docs.rs/blake3/, 2022. Online; accessed 2022.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 3–22, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[CS10] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, Heidelberg, January 2010.

[DEK20a] Anders Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. Cryptology ePrint Archive, Report 2020/1330, 2020. https://eprint.iacr.org/2020/1330.

[DEK20b] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *PoPETs*, 2020(4):355–375, October 2020.

[DFC20] Marc Peter Deisenroth, A. Aldo Faisal, and Soon Ong Cheng. *Mathematics for Machine Learning*. Cambridge University Press, 2020. https://mml-book.com.

[dna21] dnaq. Crate sodiumoxide. https://docs.rs/sodiumoxide/0.2.6/sodiumoxide/randombytes/fn.randombytes_into.html, 2021. Online; accessed 2021.

[EGK+20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Heidelberg, August 2020.

[EKR17] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3), 2017.

[Gal21] GaloisInc. Swanky. https://github.com/GaloisInc/swanky, 2021. Online; accessed 2021.

[GKWY20] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy*, pages 825–841. IEEE Computer Society Press, May 2020.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 1575–1590. ACM Press, November 2020.

[KS21] Marcel Keller and Ke Sun. Effectiveness of mpc-friendly softmax replacement, 2021.

[Lib21] Libsodium team. Libsodium. https://github.com/jedisct1/libsodium, 2021. Online; accessed 2021.

[MR18] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

[RW19] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Heidelberg, December 2019.

[WGC19] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, July 2019.

A Extended kernels

A.1 More general additive to replicated share conversion

Additive to replicated share conversion protocol.

A.1.1 $\mathsf{AdtToRep}_{\mathsf{rep}}(R; \langle \mathbf{x} \rangle_R^{\mathsf{adt}}) \to \langle \mathbf{y} \rangle^{\mathsf{rep}}$

Let $\mathsf{sync\_key}_1$ and $\mathsf{sync\_key}_2$ be distinct fixed values.

1. Let $(\mathbf{x}_1, \mathbf{x}_2) = \langle \mathbf{x} \rangle_R^{\mathsf{adt}}$.
2. Let $(A_1, A_2) \leftarrow \mathsf{adt.host\_placements}()$.
3. Let $(R_1, R_2, R_3) \leftarrow \mathsf{rep.host\_placements}()$.
4. Select one R_i such that $R_i \neq A_j, \forall j \in [2]$ which will act as the third party provider for the other $R_k, k \neq i$.
5. On $\mathsf{adt}[A_1, A_2]$:
   (a) compute $\mathsf{shape} \leftarrow \mathsf{Shape}(\langle \mathbf{x} \rangle^{\mathsf{adt}})$.
6. On A_1:
   (a) Send shape_1 to R_i.
7. On R_i:
   (a) Receive shape_1 from A_1.
   (b) $k \leftarrow \mathsf{GenPrfKey}()$.
   (c) $\mathsf{seed}_1 \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_1, k)$.
   (d) $\mathsf{seed}_2 \leftarrow \mathsf{DeriveSeed}(\mathsf{sync\_key}_2, k)$.
   (e) $\mathbf{y}_j^i \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}_1, \mathsf{seed}_j)$ for $j \in [2]$.
8. On A_1:
   (a) Receive seed_1 from R_i.
   (b) Compute $\mathbf{y}_1 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}_1, \mathsf{seed}_1)$.
9. On A_2:
   (a) Receive seed_2 from R_i.
   (b) Compute $\mathbf{y}_2 \leftarrow \mathsf{SampleUniformSeeded}(R; \mathsf{shape}_2, \mathsf{seed}_2)$.
10. Let $\langle \mathbf{y} \rangle_R^{\mathsf{adt}} = (\mathbf{y}_1, \mathbf{y}_2)$.
11. $\mathbf{c} \leftarrow \mathsf{Open}_{[A_1, A_2]} \left(\langle \mathbf{x} \rangle_R^{\mathsf{adt}} - \langle \mathbf{y} \rangle_R^{\mathsf{adt}} \right)$
12. Each A_j sends $\mathbf{y}_j$ to R_k for which $R_k = A_j \ \forall j \in [2], k \in [3]$.
13. The A_j parties that didn't have a 1-1 match to R_k in the previous step now pick an unique $R_k \neq R_i$ that hasn't received any $\mathbf{y}$ share and sends it to R_k.
14. The set of shares owned by the R parties are: $\{\{\mathbf{y}_1, \mathbf{c}\}, \{\mathbf{y}_2, \mathbf{c}\}, \{\mathbf{y}_1^i, \mathbf{y}_2^i\}\}$.
15. Parties order their shares such that it is a valid sharing of $\mathbf{y}$.

16. Return $\langle \mathbf{y} \rangle_R^{\mathsf{rep}}$.

Note that in the last steps the parties must provide a valid ordering of their shares such that it reconstructs to $\mathbf{y}$. There are multiple cases here, depending on which host placement the share provider (R_i) was assigned and the way the parties on the additive placement sent their shares to the replicated ones.

$$
\begin{aligned}
(R_i = R_1, A_1 = R_2, A_2 = R_3) &\mapsto \big((\mathbf{y}_2^1, \mathbf{y}_1^1), (\mathbf{y}_1, \mathbf{c}), (\mathbf{c}, \mathbf{y}_2)\big) \\
(R_i = R_1, A_2 = R_2, A_1 = R_3) &\mapsto \big((\mathbf{y}_1^1, \mathbf{y}_2^1), (\mathbf{y}_2, \mathbf{c}), (\mathbf{c}, \mathbf{y}_1)\big) \\
(A_1 = R_1, R_i = R_2, A_2 = R_3) &\mapsto \big((\mathbf{c}, \mathbf{y}_1), (\mathbf{y}_1^2, \mathbf{y}_2^2), (\mathbf{y}_2, \mathbf{c})\big) \\
(A_2 = R_1, R_i = R_2, A_1 = R_3) &\mapsto \big((\mathbf{c}, \mathbf{y}_2), (\mathbf{y}_2^2, \mathbf{y}_1^2), (\mathbf{y}_1, \mathbf{c})\big) \\
(A_1 = R_1, A_2 = R_2, R_i = R_3) &\mapsto \big((\mathbf{y}_1, \mathbf{c}), (\mathbf{c}, \mathbf{y}_2), (\mathbf{y}_2^3, \mathbf{y}_1^3)\big) \\
(A_2 = R_1, A_1 = R_2, R_i = R_3) &\mapsto \big((\mathbf{y}_2, \mathbf{c}), (\mathbf{c}, \mathbf{y}_1), (\mathbf{y}_1^3, \mathbf{y}_2^3)\big)
\end{aligned}
$$