Cryptographic Whitepaper

Cape Privacy

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].

Note that this paper is currently scoped to an implementation of secure linear regression. Future versions of the paper will expand upon this with additional models and protocols.

1.1 Linear regression

We focus on performing linear regression between two distinct data owners, one providing the explanatory variable X and another providing some target (or response) variable y. For simplicity, we assume X is an $n \times p$ matrix with $n > p$ and y a column vector, indexed row-wise by X_i and y_i, respectively.

Solving the classical linear regression problem equates to finding the minimizer of the mean squared error objective

$$w^* = \min_w \frac{1}{n} \sum_n (X_i \cdot w - y_i)^2$$

Using the method of maximum likelihood estimation [DFC20], we can reformulate this optimization problem into the product

$$\begin{aligned} w^* &= (X^T X)^{-1} X^T y \\ &= Z \cdot y \end{aligned}$$

Note that evaluating $Z = (X^T X)^{-1} X^T$ can be performed as local preprocessing on plaintext data by the provider of X (since the righthand term does not involve the target variable y). Thus, solving this regression problem securely reduces to a single secret-shared matrix-vector product $w = Z \cdot y$.

Additionally, we may want to compute certain metrics to assess the regression outcome. These metrics must also be evaluated on secret-shared data. Concretely, we present cryptographic protocols to support mean squared error (MSE), mean absolute percentage error (MAPE), and R-squared (R^2) metrics. Denote the target predictions $\hat{y} = X \cdot w$ and the expected target value as $\bar{y} = \frac{1}{n} \sum_n y_i$. An argument similar to the above reduces the secure computation of R^2 to securely computing the residual sum of squares (RSS):

$$\begin{aligned} R^2 &= 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2} \\ &= 1 - \frac{RSS(\hat{y}, y)}{SS(\hat{y}, y)} \end{aligned}$$

since only the RSS term depends on data from both providers (i.e. both X and y).

In summary, we are ultimately concerned with providing protocols for securely evaluating the following values:

$$
\begin{aligned}
w &= Z \cdot y \\
\hat{y} &= X \cdot w \\
MSE(\hat{y}, y) &= \frac{1}{n}\sum_{n}(\hat{y} - y_i)^2 \\
RSS(\hat{y}, y) &= \sum_{n}(\hat{y} - y_i)^2 \\
\mathsf{MAPE}(\hat{y}, y) &= \frac{1}{n}\sum_{n}\left|\frac{\hat{y} - y_i}{y_i}\right|
\end{aligned}
$$

Finally, these operations must be implemented for fixed-point numbers as described in Section 2.

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 $1 \cdot 2^{2f}$ and $\bar{x}$. 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.

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.

In some configurations we also use this pair to derive unique nonces for encrypting messages (see Section 5.3). This is done using a secure hash function as $\mathsf{nonce} = \mathsf{h}(\mathsf{sid} \| \mathsf{rdv\_key})$ and security again depends on sid having fixed length ℓ.

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 rdv_key attribute of Send and Receive nodes in the graph being inlined must be updated to h(ac_key||rdv_key) where ac_key is the activation key of the Enter and Exit nodes being replaced, and rdv_key in the graph being inlined into must be updated to h(rdv_key). Other examples are the sync_key attribute of DeriveSeed operations and the ac_key attribute of Enter and Exit operations. For this to be secure we require all ac_key, rdv_key, and 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. We currently only present protocols for the operations needed to support linear regressions and associated metrics (Section 1.1).

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 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].

4.3 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 2. Subtraction as shown in Figure 3 is almost identical, with the parties simply subtracting their shares locally instead of adding. Correctness and security follow from [ABF+16].

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

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 $\mathbf{k} = \big((k_1^1, k_2^1), (k_2^2, k_3^2), (k_3^3, k_1^3)\big)$.

Fig. 1. Replicated setup.

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

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 $\langle \mathbf{z} \rangle = \big((\mathbf{z}_1^1, \mathbf{z}_2^1), (\mathbf{z}_2^2, \mathbf{z}_3^2), (\mathbf{z}_3^3, \mathbf{z}_1^3)\big)$.

Fig. 2. Replicated addition.

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

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 $\langle \mathbf{z} \rangle = \big((\mathbf{z}_1^1, \mathbf{z}_2^1), (\mathbf{z}_2^2, \mathbf{z}_3^2), (\mathbf{z}_3^3, \mathbf{z}_1^3)\big)$.

Fig. 3. Replicated subtraction.

4.4 Multiplication

Protocol RepMul in Figure 4 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 ZeroShare protocol in Figure 5, and inherit their correctness and security proofs.

As for the ZeroShare protocol, note that the DeriveSeed operation is parameterized by an explicit 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 [Lib21] (BLAKE2b) with an output truncated to 128 bits; this ensures that all seeds are unique across sessions when sid and sync_key are of fixed length ℓ. Finally, the SampleUniformSeeded operation is implemented by running AES as a PRNG using the seed as the key and encrypting plaintexts $0, 1, ...$ in ECB mode.

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

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

Fig. 4. Replicated multiplication

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

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 $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \boldsymbol{\alpha}_3)$.

Fig. 5. Zero share protocol.

4.5 Sum and Mean

Similarly to addition, we build a summation protocol (Figure 6) 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.

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

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 $\langle \mathbf{z} \rangle = \big((\mathbf{z}_1^1, \mathbf{z}_2^1), (\mathbf{z}_2^2, \mathbf{z}_3^2), (\mathbf{z}_3^3, \mathbf{z}_1^3)\big)$.

Fig. 6. Replicated summation.

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 the correct dimensions. For more details, the reader can check the `replicated_dot_product` definition in the compiler.

4.7 Sharing data

In Figure 7 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.

4.8 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 8) or to all parties (Figure 9). Note that Figure 8 corresponds to a replicated opening, whereas Figure 9 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.

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

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

Fig. 7. Replicated sharing protocol.

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

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 $\mathbf{x}$.

Fig. 8. Replicated opening protocol to a specific party P_i.

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

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 $\mathbf{x}^2$.

Fig. 9. Additive opening protocol.

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 our semi-honest three-parties model we implement the probabilistic truncation from [DEK20b] which avoids preprocessing.

The protocol is described in Fig 10. At a high-level the idea is to convert the replicated sharing to a $(2, 2)$ additive sharing and then execute the truncation protocol between two parties (Figure 11). The preprocessing for the 2PC computation is generated by P_3 which is later additively shared to P_1 and P_2 (Figure 12). After the two-party truncation is done, the output share is then converted back to a replicated sharing (Figure 14). Correctness and security follow from [DEK20b].

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] The main idea is for each party to locally bit-decompose 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 $\mathbb{Z}_{2^k}$. In ABY3 this was done using a three-party OT protocol. In MP-SPDZ [Kel20] the boolean to ring sharing conversion was achieved using a daBit.
2. SecureNN [WGC19] similar to ABY3 with the downside that it uses arithmetic modulo some small fields but avoids ring-to-boolean conversion.
3. Comparisons using edaBits [EGK+20]. The 3PC case has roughly the same cost as ABY3.

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 had along with edaBits preprocessing by introducing our custom protocol B2A for binary to ring share conversion in Figure 17. The

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

1. Let $\left((\mathbf{x}_1^1, \mathbf{x}_2^1), (\mathbf{x}_2^2, \mathbf{x}_3^2), (\mathbf{x}_3^3, \mathbf{x}_1^3)\right) = \langle \mathbf{x} \rangle^{\mathsf{rep}}_{2^k}$.
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 $\langle \mathbf{y} \rangle^{\mathsf{rep}} \leftarrow \mathsf{AdtToRep}_{[P_1,P_2,P_3]}(\langle \mathbf{y}' \rangle^{\mathsf{adt}})$.

Fig. 10. Three party truncation protocol.

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

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 $\langle \mathbf{y} \rangle \leftarrow \langle \mathbf{y}_+ \rangle - 2^{k-m-1}$.

Fig. 11. Two-party truncation protocol.

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

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 $\langle \mathbf{x} \rangle^{\mathsf{adt}} = (\mathbf{x}_1^1, \mathbf{x}_2^2)$.

Fig. 12. Additive sharing.

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

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 $\langle \mathbf{x}' \rangle^{\mathsf{adt}} = (\mathbf{x}'_1, \mathbf{x}'_2)$.

Fig. 13. Conversion from replicated shares to additive shares.

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

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 $\langle \mathbf{y} \rangle_R^{\mathsf{rep}} = \left((\mathbf{y}_1, \mathbf{c}), (\mathbf{c}, \mathbf{y}_2), (\mathbf{y}_3^3, \mathbf{y}_1^3)\right)$.

Fig. 14. Additive to replicated share conversion protocol.

MSB protocol can be found in Figure 16 where $\overline{\cdot}$ is used to denote a vector with k elements indexed using (i) for $i \in [k]$.

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

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

Fig. 15. Ring bit decomposition to binary shares

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

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

Fig. 16. MSB computation from a replicated ring share

Improved ABY3 boolean to ring sharing protocol

When the inputs are tensors we can perform the share conversion $\langle \cdot \rangle_2 \mapsto \langle \cdot \rangle_{2^k}$ ($\mathsf{B2A}$ 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. We fully describe this share conversion protocol in Figure 17.

Acting as a trusted third party, P_3 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. Then 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 $\mathbf{c} \leftarrow \mathbf{x}_2 \oplus \mathbf{b}_2$ and then locally XOR-ing in the arithmetic domain $\mathbf{x}_{2^k} = \mathbf{c} + \mathbf{b}_{2^k} - 2 \cdot \mathbf{c} \cdot \mathbf{b}_{2^k}$. Finally they convert back to a replicated sharing using $\mathsf{AdtToRep}$ from Figure 14. Note that in our implementation of the additive to replicated share conversions we consider a more general variant where $\langle \mathbf{x} \rangle^{\mathsf{adt}}$ is converted to $\langle \mathbf{x} \rangle^{\mathsf{rep}}$ where the host placements on the additive placement adt are not necessarily a subset of host placements composing rep given in Figure 28.

5 Implementation

5.1 Runtime

Operations on ring tensors is 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.

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

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

Fig. 17. Binary to arithmetic share conversion computation.

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

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) $\mathbf{b}_{2^k}^1 \leftarrow \mathsf{SampleUniformSeeded}(\mathbb{Z}_{2^k}; \mathsf{seed}_{2^k}, \mathsf{shape})$.
   (j) $\mathbf{b}_{2^k}^2 \leftarrow \mathbf{b} - \mathbf{b}_{2^k}^1$.
   (k) Send $\mathbf{b}_2^2$ and $\mathbf{b}_{2^k}^2$ to P_2.
2. On P_1:
   (a) Receive seed_2 and seed_{2^k} from D.
   (b) $\mathbf{b}_2^1 \leftarrow \mathsf{SampleUniformSeeded}(\mathbb{Z}_2; \mathsf{seed}_2, \mathsf{shape})$.
   (c) $\mathbf{b}_{2^k}^1 \leftarrow \mathsf{SampleUniformSeeded}(\mathbb{Z}_{2^k}; \mathsf{seed}_{2^k}, \mathsf{shape})$.
3. On P_2:
   (a) Receive $\mathbf{b}_2^2$ and $\mathbf{b}_{2^k}^2$ from D.
4. Let $\langle \mathbf{b}\rangle_2^{\mathsf{adt}} = (\mathbf{b}_2^1, \mathbf{b}_2^2)$.
5. Let $\langle \mathbf{b}\rangle_{2^k}^{\mathsf{adt}} = (\mathbf{b}_{2^k}^1, \mathbf{b}_{2^k}^2)$
6. Return $\langle \mathbf{b}\rangle_2^{\mathsf{adt}}$ and $\langle \mathbf{b}\rangle_{2^k}^{\mathsf{adt}}$.

Fig. 18. DaBit protocol generation

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

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

Fig. 19. Absolute value computation from a replicated ring share.

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].

5.3 Key Setup Infrastructure (Work In Progress)

The scope of this subsection is to detail how workers can establish secure channels between themselves without trusting `Cape` and without relying on TLS CA infrastructure. Concurrent to our writing, a blogpost appeared on establishing keys between MPC parties over the internet [Ome21].

For our linear regression use case there are three organizations: `A`, `B` and `Cape`. Each organization owns a set of workers to perform computations on encrypted data. In this context, workers are the parties running the MPC protocol. On both sides of `A` and `B` there are two different types of parties (besides the workers): an operator and a data scientist denoted as `OpA`, `DS-A`, `OpB`, and `DS-B` respectively. The operators and data scientists talk to each other across organizations through a communication service ran by `Cape`.

We now describe the key steps for workers to establish authenticated secret channels in order for the replicated computations to be done securely:

1. `OpA` generates a signature key pair $\mathsf{vk}_A, \mathsf{sk}_A$ using Libsodium's `crypto_sign_keypair`.
2. The secret key sk_A is made available to the workers inside organization `A` and a hash of the public verification key vk_A denoted as $h_A^{\mathsf{op}} = \mathsf{hash}(\mathsf{vk}_A)$ is sent to `B-DS` by the operator `OpA` through `Cape`.
3. Optionally, `B-DS` can check via an authenticated channel (say `mail.google.com` or some QR code using Signal app) that the hash h_A^{op} received from `Cape` corresponds to the one sent by `OpA`.
4. On the `B` side the operator `OpB` generates a symmetric key K_B using `crypto_auth_keygen` in Libsodium. Then K_B is broadcasted to all parties on the `B` side (workers and data scientist). The DS-B fetches h_A^{op} from `Cape` and sends $\tau_{h_A}^{\mathsf{DSB}} \leftarrow \mathsf{MAC}(K_B, h_A^{\mathsf{op}})$ along with $h_A^{\mathsf{DSB}} \leftarrow h_A^{\mathsf{op}}$ to all workers on the B side through `Cape` service. This MAC is used by workers on the B side to check that the public key setup messages below between workers across organizations are forwarded correctly by `Cape`.
5. On the `A` side all workers W_A^i generate an ephemeral encryption key ek_A^i and its corresponding decryption key dk_A^i using `crypto_box_keypair`.
6. For the key-setup between workers, on the `A` side each worker sends the encryption key ek_A^i, the verification key vk_A, a timestamp `datetime` and a signature $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_A, \mathsf{ek}_A^i || \texttt{datetime})$ to `Cape`. These are then forwarded by `Cape` to every worker inside organization `B`. Finally the workers on the `B`-side receive the following: $M_{\mathsf{vk}}, \mathsf{ek}_A^i, \mathsf{vk}_A, \texttt{datetime}, \sigma_i^A$. They continue if and only if the following equations hold:
   (a) $\mathsf{hash}(\mathsf{vk}_A) \stackrel{?}{=} h_A^{\mathsf{DSB}}$.
   (b) $\mathsf{Ver}(K_B, \tau_{h_A^{\mathsf{DSB}}}) \stackrel{?}{=} 1$. This ensures that the workers received the correct $\mathsf{hash}(\mathsf{vk}_A)$ sent by `B-DS` through `Cape`.
   (c) $\mathsf{Ver}(\mathsf{vk}_A, \sigma_A^i) \stackrel{?}{=} 1$ using the vk_A received from the worker through `Cape`.
   (d) If all checks pass then keep ek_A^i as the public key for worker W_A^i. Note that `datetime` is used to prevent replay attacks, this way the workers register the other PKs iff `datetime` is within some specific time frame.
7. Repeat this for parties on the B side (see Figure 27).

After all workers agree on the public key setup infrastructure using the description above, they continue by calling `crypto_box_easy` which launches the Curve25519 key exchange. The workers communicate further using a symmetric key as described in the first section. For more details on the Libsodium API the reader can refer to Figure 25. Note that communication between workers happens through an entity called `Broker`, which is simply forwarding messages between workers. Since `crypto_box_easy` authenticates the ciphertexts, they are immune to replay attacks.

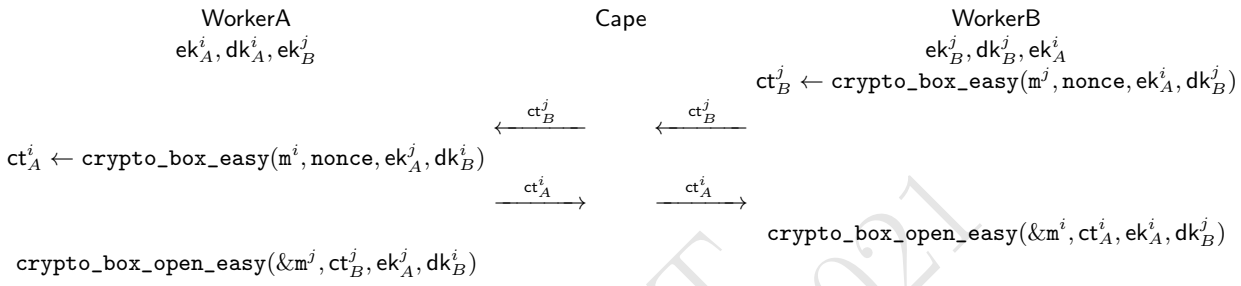

Fig. 20. Communication flow between workers

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.

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.

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.

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.

Ome21. Omer Shlomovits. Mpc-over-signal. https://medium.com/zengo/mpc-over-signal-977db599de66, 2021. Online; accessed 2021.

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.

Fig. 21. PK setup for Organization A

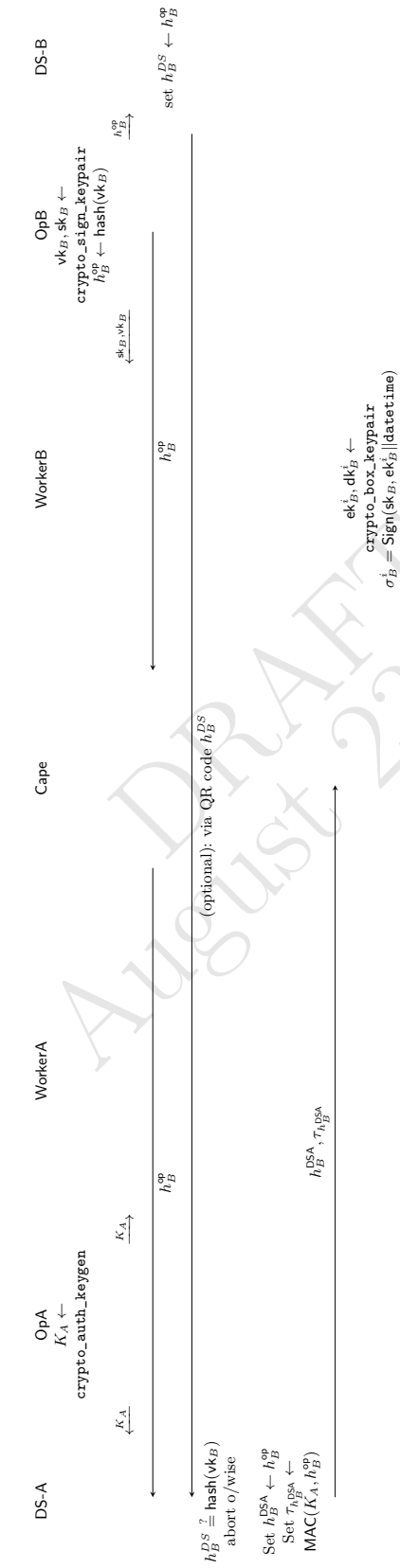

Fig. 22. PK setup for Organization B

A Extended kernels

A.1 More general additive to replicated share conversion

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

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

Fig. 23. Additive to replicated share conversion protocol.

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.

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