Don't Eject the Impostor: Fast Three-Party Computation With a Known Cheater (Supplementary Material)

1. Gates for Machine Learning

Here, for completeness we provide additional gates for fixed-point truncation and ReLUs for ASTRA [1], [4] and SWIFT [2], [4] as well as their modifications for AUXILIATOR and SOCIUM. These are the versions that we use in our communication evaluation as well as our implementation. Note that among different publications, details may slightly deviate. We use the most current versions provided by [4] with minor tweaks.

Notation: By x^d we denote the truncation of x, i.e., an arithmetic right shift by $d = 16$. By $(b)^{\mathsf{a}}$ for a bit b, we denote its representation as an element of $\mathbb{Z}_{2^\ell}$, i.e., its padding by $\ell-1 = 63$ 0 bits. To clearly distinguish between shares in the binary and arithmetic domains, we use, e.g., $\langle x\rangle$ for an arithmetic sharing of $x \in \mathbb{Z}_{2^\ell}$ and $\langle x\rangle^{\mathbf{B}}$ for a binary sharing of $b \in \mathbb{Z}_2$. By $x[i]$, we denote bit i of integer x, i.e., x_i where x is decomposed into x_j with $x = \sum_{j=0}^{\ell-1} x_j$.

1.1. Truncation Pair Generation

SWIFT requires a designated protocol to generate truncation pairs that is used in further operations.

Protocol $\Pi_{\mathsf{Tr}}()$

Input: None.

Output: $\langle r\rangle$ for random $r \in \mathbb{Z}_{2^\ell}$, and $[\![r^d]\!]$.

Offline:

1. For $s \in \{1,2\}, 0 \le i < \ell$, $\mathsf{S}_0, \mathsf{S}_s$ non-interactively sample random bit $r_{s,i} \in \{0,1\}$.
2. Compute $\langle r\rangle$:
   a. Compute $\langle r_{s,i}\rangle$ for $s \in \{1,2\}, 0 \le i < \ell$ by setting $r^0_{1,i} = r^2_{1,i} = r^0_{2,i} = r^1_{2,i} = 0 \in \mathbb{Z}_{2^\ell}$ and $r^1_{1,i} = (r_{1,i})^{\mathsf{a}}, r^2_{2,i} = (r_{2,i})^{\mathsf{a}}$ for $0 \le i < \ell$.
   b. Set up vector $\langle \vec{\mathbf{A}}\rangle$ of dimension ℓ where $\langle \vec{\mathbf{A}}_{i+1}\rangle = 2^{i+1} \cdot \langle r_{1,i}\rangle$ for $0 \le i < \ell$.
   c. Set up vector $\langle \vec{\mathbf{B}}\rangle$ of dimension ℓ where $\langle \vec{\mathbf{B}}_{i+1}\rangle = \langle r_{2,i}\rangle$ for $0 \le i < \ell$.
   d. Invoke $\mathcal{F}_{\mathsf{MultPre}}$ on $\langle \vec{\mathbf{A}}\rangle^{\intercal}$ and $\langle \vec{\mathbf{B}}\rangle$ to obtain $\langle x\rangle$ with $x = \vec{\mathbf{A}}^{\intercal} \cdot \vec{\mathbf{B}}$.
   e. Locally compute $\langle r\rangle = \sum_{i=0}^{\ell-1} 2^i(\langle r_{1,i}\rangle + \langle r_{2,i}\rangle) - \langle x\rangle$.
3. Compute $[\![r^d]\!]$:
   a. Compute $\langle r_{s,i}\rangle$ for $s \in \{1,2\}, f \le i < \ell$ by setting $\lambda^0_{r_{1,i}} = \lambda^2_{r_{1,i}} = \mathsf{m}_{r_{1,i}} = \lambda^0_{r_{2,i}} = \lambda^1_{r_{2,i}} = \mathsf{m}_{r_{2,i}} = 0 \in \mathbb{Z}_{2^\ell}$ and $\lambda^1_{r_{1,i}} = (r_{1,i})^{\mathsf{a}}, \lambda^2_{r_{2,i}} = (r_{2,i})^{\mathsf{a}}$ for $f \le i < \ell$.
   b. Set up vector $\langle \vec{\mathbf{C}}\rangle$ of dimension $\ell - f$ where $\langle \vec{\mathbf{C}}_{i+1}\rangle = 2^{i+1} \cdot \langle r_{1,f+i}\rangle$ for $0 \le i < \ell-f-1$ and $\langle \vec{\mathbf{C}}_{\ell-f}\rangle = (\sum_{i=\ell-f}^{\ell} 2^i)\langle r_{1,\ell-1}\rangle$.
   c. Set up vector $\langle \vec{\mathbf{D}}\rangle$ of dimension $\ell - f$ where $\langle \vec{\mathbf{D}}_{i+1}\rangle = \langle r_{2,f+i}\rangle$ for $0 \le i < \ell - f$.
   d. Invoke $\mathcal{F}_{\mathsf{MultPre}}$ on $\langle \vec{\mathbf{C}}\rangle^{\intercal}$ and $\langle \vec{\mathbf{D}}\rangle$ to obtain $\langle y\rangle$ with $y = \vec{\mathbf{C}}^{\intercal} \cdot \vec{\mathbf{D}}$.
   e. Locally compute $\langle r^d\rangle = \sum_{i=0}^{\ell-f-2} 2^i(\langle r_{1,f+i}\rangle + \langle r_{2,f+i}\rangle) + \sum_{i=\ell-f-1}^{\ell-1} 2^i(\langle r_{1,\ell-1}\rangle + \langle r_{2,\ell-1}\rangle) - \langle y\rangle$ and set $\mathsf{m}_{r^d} = 0$ to obtain $[\![r^d]\!]$.

Figure 1: SWIFT: Truncation pair generation protocol.

1.2. Matrix Multiplication With Truncation

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{Mult}}(\langle \vec{\mathbf{X}}\rangle, \langle \vec{\mathbf{Y}}\rangle)$

Input: $\langle\cdot\rangle$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}^{u\times w}_{2^\ell}$ and $\vec{\mathbf{Y}} \in \mathbb{Z}^{w\times v}_{2^\ell}$.

Output: $\langle\cdot\rangle$-shares of $\vec{\mathbf{Z}} \in \mathbb{Z}^{u\times v}_{2^\ell}$ with $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}}$.

Setup:

1. Non-interactively generate $[\vec{\mathbf{Q}}]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\vec{\mathbf{Q}}^{\mathbf{1}} \in \mathbb{Z}^{u\times v}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\vec{\mathbf{Q}}^{\mathbf{2}} \in \mathbb{Z}^{u\times v}_{2^\ell}$.
2. $\mathsf{S}_0, \mathsf{S}_1$ sample random $\vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}} \in \mathbb{Z}^{u\times v}_{2^\ell}$.
3. S_0 computes and sends $\vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}} = \left((\vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}) - \vec{\mathbf{Q}}^{\mathbf{1}} - \vec{\mathbf{Q}}^{\mathbf{2}}\right)^d - \vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}}$ to S_2.

Online:

1. Locally compute the following:
   – S_1: $\vec{\mathbf{P}}^{\mathbf{1}} = (\vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}}) + (\vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}) + \vec{\mathbf{Q}}^{\mathbf{1}}$.
   – S_2: $\vec{\mathbf{P}}^{\mathbf{2}} = (\vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}}) + (\vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}) + \vec{\mathbf{Q}}^{\mathbf{2}}$.
2. $\mathsf{S}_1, \mathsf{S}_2$ exchange $\vec{\mathbf{P}}^{\mathbf{1}}, \vec{\mathbf{P}}^{\mathbf{2}}$ and reconstruct $\vec{\mathsf{m}}_{\mathbf{Z}} = \left(\vec{\mathbf{P}}^{\mathbf{1}} + \vec{\mathbf{P}}^{\mathbf{2}} + (\vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}})\right)^d$.

Figure 2: ASTRA: Matrix multiplication protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{Mult}}(\langle \vec{\mathbf{X}}\rangle, \langle \vec{\mathbf{Y}}\rangle)$

Input: $\langle\cdot\rangle$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}^{u\times w}_{2^\ell}, \vec{\mathbf{Y}} \in \mathbb{Z}^{w\times v}_{2^\ell}$.

Output: $\langle\cdot\rangle$-shares of $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} \in \mathbb{Z}^{u\times v}_{2^\ell}$.

Setup:

1. Invoke $\mathcal{F}_{\mathsf{MultPre}}([\vec{\lambda}_{\mathbf{X}}], [\vec{\lambda}_{\mathbf{Y}}])$ to obtain $[\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}]$.
2. Non-interactively generate $[\vec{\lambda}_{\mathbf{Z}}]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}} \in \mathbb{Z}^{u\times v}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}} \in \mathbb{Z}^{u\times v}_{2^\ell}$.

Online:

1. Locally compute the following:
   - S_1: $\vec{\mathbf{P}}^{\mathbf{1}} = \left((\vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}) + (\vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}}) + (\vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}) + \vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}\right)^d - \vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}}$.
   - S_2: $\vec{\mathbf{P}}^{\mathbf{2}} = \left((\vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}}) + (\vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}) + \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}}\right)^d - \vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}}$.
2. $\mathsf{S}_1, \mathsf{S}_2$ exchange $\vec{\mathbf{P}}^{\mathbf{1}}, \vec{\mathbf{P}}^{\mathbf{2}}$ and reconstruct $\vec{\mathsf{m}}_{\mathbf{Z}} = \vec{\mathbf{P}}^{\mathbf{1}} + \vec{\mathbf{P}}^{\mathbf{2}}$.

Figure 3: AUXILIATOR: Matrix multiplication protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{Mult}}([\![\vec{\mathbf{X}}]\!], [\![\vec{\mathbf{Y}}]\!])$

Input: $[\![\cdot]\!]$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}_{2^\ell}^{u \times w}, \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{w \times v}$.

Output: $[\![\cdot]\!]$-shares of $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

Setup:

1. Invoke $\mathcal{F}'_{\mathsf{MultPre}}$ on $\langle \vec{\lambda}_{\mathbf{X}} \rangle$ and $\langle \vec{\lambda}_{\mathbf{Y}} \rangle$ to obtain $\langle \vec{\gamma}_{\mathbf{XY}} \rangle'$ with $\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}$ (S_0 has $\vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}}, \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}}$,$\mathsf{S}_1$ has $\vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}, \vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}}$,$\mathsf{S}_2$ has $\vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}}$).
2. Non-interactively generate $\langle \vec{\lambda}_{\mathbf{Z}} \rangle$ by $\mathsf{S}_i, \mathsf{S}_{i+1}$ sampling random $\vec{\lambda}^{\mathbf{i}}_{\mathbf{Z}} \in \mathbb{Z}_{2^\ell}$ for $i \in \{0, 1, 2\}$.

Online:

1. Locally compute the following (S_0 delays this step until final verification):
   - S_1 : $\vec{\mathbf{P}}^{\mathbf{0}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{0}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{0}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}}$.
   - S_1 : $\vec{\mathbf{P}}^{\mathbf{1}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}$.
   - $\mathsf{S}_2, \mathsf{S}_0$: $\vec{\mathbf{P}}^{\mathbf{2}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}} + \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}$.
2. Send the following messages:
   - S_1 sends $\vec{\mathbf{Q}}^{\mathbf{1}} = \left(\vec{\mathbf{P}}^{\mathbf{0}} + \vec{\mathbf{P}}^{\mathbf{1}}\right)^d - \vec{\lambda}^{\mathbf{0}}_{\mathbf{Z}} - \vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}}$ to S_2.
   - S_2 sends $\vec{\mathbf{Q}}^{\mathbf{2}} = \left(\vec{\mathbf{P}}^{\mathbf{2}}\right)^d - \vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}}$ to S_1.
3. $\mathsf{S}_1, \mathsf{S}_2$ reconstruct $\vec{\mathsf{m}}_{\mathbf{Z}} = \vec{\mathbf{Q}}^{\mathbf{1}} + \vec{\mathbf{Q}}^{\mathbf{2}}$.

Final Verification (batched over all multiplications):

1. S_1 sends $\vec{\mathbf{Q}}^{\mathbf{1}}$ to S_0.
2. S_0 computes $\vec{\mathbf{Q}}^{\mathbf{2}}$ and reconstructs $\vec{\mathsf{m}}_{\mathbf{Z}} = \vec{\mathbf{Q}}^{\mathbf{1}} + \vec{\mathbf{Q}}^{\mathbf{2}}$.
3. S_0 sends $\mathsf{H}(\vec{\mathbf{Q}}^{\mathbf{2}})$ to S_1 that then checks consistency to the $\vec{\mathbf{Q}}^{\mathbf{2}}$ it received from S_2.

Figure 4: SOCIUM: Matrix multiplication protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{Mult}}([\![\vec{\mathbf{X}}]\!], [\![\vec{\mathbf{Y}}]\!])$

Input: $[\![\cdot]\!]$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}_{2^\ell}^{u \times w}, \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{w \times v}$.

Output: $[\![\cdot]\!]$-shares of $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

Setup:

1. Invoke $\mathcal{F}_{\mathsf{MultPre}}$ on $\langle \vec{\lambda}_{\mathbf{X}} \rangle$ and $\langle \vec{\lambda}_{\mathbf{Y}} \rangle$ to obtain $\langle \vec{\gamma}_{\mathbf{XY}} \rangle$ with $\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}$.
2. Non-interactively generate $\langle \vec{\lambda}_{\mathbf{W}} \rangle$ where $\mathsf{S}_1, \mathsf{S}_2$ know all shares by $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ sampling random $\vec{\lambda}^{\mathbf{0}}_{\mathbf{W}}, \vec{\lambda}^{\mathbf{2}}_{\mathbf{W}} \in \mathbb{Z}_{2^\ell}^{u \times v}$ and $\mathsf{S}_1, \mathsf{S}_2$ sampling random $\vec{\lambda}^{\mathbf{0}}_{\mathbf{W}}, \vec{\lambda}^{\mathbf{2}}_{\mathbf{W}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.
3. Obtain $\langle \vec{\mathbf{R}} \rangle$ and $[\![\vec{\mathbf{R}}^d]\!]$ for random $\vec{\mathbf{R}} \in \mathbb{Z}_{2^\ell}^{u \times v}$ by calling uv instances of Π_{Tr}.

Note: $\vec{\lambda}_{\mathbf{Z}} = \vec{\lambda}_{\mathbf{W}} + \vec{\lambda}_{\mathbf{R^d}}$.

Online:

1. Locally compute the following (S_0 delays this step until final verification):
   - $\mathsf{S}_0, \mathsf{S}_1$: $\vec{\mathbf{P}}^{\mathbf{0}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{0}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{0}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}} - \vec{\mathbf{R}}^{\mathbf{0}}$.
   - $\mathsf{S}_1, \mathsf{S}_2$: $\vec{\mathbf{P}}^{\mathbf{1}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}} - \vec{\mathbf{R}}^{\mathbf{1}}$.
   - $\mathsf{S}_2, \mathsf{S}_0$: $\vec{\mathbf{P}}^{\mathbf{2}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}} - \vec{\mathbf{R}}^{\mathbf{2}}$.
2. $\mathsf{S}_1, \mathsf{S}_2$ exchange $\vec{\mathbf{P}}^{\mathbf{0}}, \vec{\mathbf{P}}^{\mathbf{2}}$, reconstruct $\vec{\mathbf{P}} = \vec{\mathbf{P}}^{\mathbf{0}} + \vec{\mathbf{P}}^{\mathbf{1}} + \vec{\mathbf{P}}^{\mathbf{2}}$ and set $\vec{\mathbf{W}} = \left(\vec{\mathbf{P}} + \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}\right)^d$.
3. $\mathsf{S}_1, \mathsf{S}_2$ set up $[\![\vec{\mathbf{W}}]\!]$ by setting $\vec{\mathsf{m}}_{\mathbf{W}} = \vec{\mathbf{W}} - (\vec{\lambda}^{\mathbf{0}}_{\mathbf{W}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{W}} + \vec{\lambda}^{\mathbf{2}}_{\mathbf{W}})$
4. $\mathsf{S}_1, \mathsf{S}_2$ compute $[\![\vec{\mathbf{Z}}]\!] = [\![\vec{\mathbf{W}}]\!] + [\![\vec{\mathbf{R}}^d]\!]$.

Final Verification (batched over all multiplications):

1. S_1 sends $\vec{\mathsf{m}}_{\mathbf{W}}$ and S_2 sends $\mathsf{H}(\vec{\mathsf{m}}_{\mathbf{W}})$ to S_0 that then checks consistency.
2. S_0 computes $\vec{\mathbf{P}}^{\mathbf{0}}, \vec{\mathbf{P}}^{\mathbf{2}}$ and sends $\mathsf{H}(\vec{\mathbf{P}}^{\mathbf{2}})$ to S_1 and $\mathsf{H}(\vec{\mathbf{P}}^{\mathbf{0}})$ to S_2 that then check consistency to the values received in online step 2.
3. S_0 computes its shares of $[\![\vec{\mathbf{Z}}]\!]$.

Figure 5: SWIFT: Matrix multiplication protocol for fixed-point arithmetic.

1.3. Multiplication by Constant With Truncation

For multiplication with a non-integer constant, truncation is required rendering the operation interactive.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{MultConstant}}(\langle x \rangle, c)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$, and a public constant $c \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = c \cdot x$.

Setup:

1. Generate $[\lambda_z]$ with $\lambda_z = c \cdot \lambda_x$:
   - $\mathsf{S}_0, \mathsf{S}_1$ sample random $\lambda^1_z \in \mathbb{Z}_{2^\ell}$.
   - S_0 computes and sends $\lambda^2_z = (c \cdot \lambda_x)^d - \lambda^1_z$ to S_2.

Online:

2. $\mathsf{S}_1, \mathsf{S}_2$ compute $\mathsf{m}_z = (c \cdot \mathsf{m}_x)^d$.

Figure 6: ASTRA: Multiplication with a constant protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{MultConstant}}(\langle x \rangle, c)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$, and a public constant $c \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = c \cdot x$.

Setup:

1. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda^1_z \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda^2_z \in \mathbb{Z}_{2^\ell}$.
2. S_2 computes and sends $p^2 = \left(c \cdot \lambda^2_x\right)^d - \lambda^2_z$ to S_1.

Online:

1. S_1 computes and sends $p^1 = \left(c \cdot (\mathsf{m}_x + \lambda^1_x)\right)^d - \lambda^1_z$ to S_2.
2. $\mathsf{S}_1, \mathsf{S}_2$ compute $\mathsf{m}_z = p^1 + p^2$.

Figure 7: AUXILIATOR: Multiplication with a constant protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{MultConstant}}(\llbracket x \rrbracket, c)$

Input: $\llbracket \cdot \rrbracket$-shares of $x \in \mathbb{Z}_{2^\ell}$ and a public constant $c \in \mathbb{Z}_{2^\ell}$.

Output: $\llbracket \cdot \rrbracket$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = c \cdot x$.

Setup:

1. Non-interactively generate $\langle \lambda_z \rangle$ by $\mathsf{S}_i, \mathsf{S}_{i+1}$ sampling random $\lambda_z^i \in \mathbb{Z}_{2^\ell}$ for $i \in \{0, 1, 2\}$.
2. S_0 computes and sends $p^0 = \left(c(\lambda_x^0 + \lambda_x^2)\right)^d - \lambda_z^0$ to S_2.
3. S_0 computes and sends $p^2 = \left(c(\lambda_x^0 + \lambda_x^2)\right)^d - \lambda_z^2$ to S_1.

Online:

1. $\mathsf{S}_1, \mathsf{S}_2$ compute $p^1 = \left(c(\mathsf{m}_x + \lambda_x^1)\right)^d - \lambda_z^1$.
2. $\mathsf{S}_1, \mathsf{S}_2$ compute m_z:
   - S_1 computes $\mathsf{m}_z = p^1 + p^2 - \lambda_z^0$.
   - S_2 computes $\mathsf{m}_z = p^1 + p^0 - \lambda_z^2$.

Final Verification (batched over all multiplications):

1. S_1 sends p^1 to S_0.
2. S_0 computes m_z like S_1 did in online step 2.

Figure 8: SOCIUM: Multiplication with a constant protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{MultConstant}}(\llbracket x \rrbracket, c)$

Input: $\llbracket \cdot \rrbracket$-shares of $x \in \mathbb{Z}_{2^\ell}$ and a public constant $c \in \mathbb{Z}_{2^\ell}$.

Output: $\llbracket \cdot \rrbracket$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = c \cdot x$.

Setup:

1. Obtain $\langle r \rangle$ and $\llbracket r^d \rrbracket$ for random $r \in \mathbb{Z}_{2^\ell}$ by calling Π_{Tr}.
2. Reinterpret $\langle r \rangle$ as $\llbracket r \rrbracket$ by setting $\mathsf{m}_r = 0$.

Online:

1. $\mathsf{S}_1, \mathsf{S}_2$ compute $\llbracket p \rrbracket = c \cdot \llbracket x \rrbracket - \llbracket r \rrbracket$.
2. $\mathsf{S}_1, \mathsf{S}_2$ reconstruct p:
   - $\mathsf{S}_1, \mathsf{S}_2$ exchange λ_p^0, λ_p^2 and set $p = \mathsf{m}_p + \lambda_p^0 + \lambda_p^1 + \lambda_p^2$.
3. $\mathsf{S}_1, \mathsf{S}_2$ compute $\llbracket z \rrbracket = p^d + \llbracket r^d \rrbracket$.

Final Verification (batched over all multiplications):

1. S_1 sends λ_p^1 and S_2 sends $\mathsf{H}(\lambda_p^1)$ to S_0 that then checks consistency.
2. S_0 computes p like $\mathsf{S}_1, \mathsf{S}_2$ did in online step 2.
3. S_0 sends $\mathsf{H}(\lambda_p^2)$ to S_1 and $\mathsf{H}(\lambda_p^0)$ to S_2 that then check consistency to the values received in online step 2.
4. S_0 computes its shares of $\llbracket z \rrbracket$.

Figure 9: SWIFT: Multiplication with a constant protocol for fixed-point arithmetic.

1.4. Rectified Linear Units (ReLUs)

Note that our ReLUs work for integer and fixed-point arithmetic, but internally only utilize integer arithmetic. This is the case as the functionality is exactly the same for both cases: The most significant bit (MSB) always returns the sign of the number and multiplication by an integer sharing of 0 or 1 without truncation to another integer or fixed-point number is correct. The ReLU protocol is generic and depends on instantiations of Π_{MSB} and Π_{BitA} discussed in subsequent sections.

Protocol $\Pi_{\mathsf{ReLU}}(\langle v \rangle)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = \max(0, x)$.

Online:

1. $\langle s \rangle^{\mathbf{B}} = \Pi_{\mathsf{MSB}}(\langle x \rangle)$.
2. $\langle t \rangle^{\mathbf{B}} = \langle s \rangle^{\mathbf{B}} \oplus \langle 1 \rangle^{\mathbf{B}}$.
3. $\langle t \rangle = \Pi_{\mathsf{BitA}}(\langle t \rangle^{\mathbf{B}})$.
4. $\langle z \rangle = \langle t \rangle \cdot \langle x \rangle$.

Figure 10: All protocols: Rectified linear unit (ReLU) protocol. For use in a specific protocol, it is sufficient to replace $\langle \cdot \rangle$-shares by the protocol's sharings and to then use the respective gates.

1.5. Most Significant Bit (MSB) Extraction

Like ASTRA and SWIFT, we utilize parallel prefix adders (PPAs) for MSB extraction. The utilized PPA (provided alongside our implementation code) uses 128 AND gates and has a multiplicative depth of 6 for $\ell = 64$.

Protocol $\Pi_{\mathsf{MSB}}(\langle x \rangle)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle^{\mathbf{B}}$-shares of $s \in \mathbb{Z}_2$ with $s = 1$ iff $x < 0$.

Setup:

1. $\mathsf{S}_0, \mathsf{S}_1$ sample random $\lambda_{a_k}^1 \in \mathbb{Z}_2$ for $0 \leq k < \ell$.
2. S_0 computes and sends $\lambda_{a_k}^2 = (\lambda_x^1 + \lambda_x^2)[k] \oplus \lambda_{a_k}^1$ to S_2 for $0 \leq k < \ell$.
2. Set up $\langle a_k \rangle^{\mathbf{B}} = \langle (\lambda_x^1 + \lambda_x^2)[k] \rangle^{\mathbf{B}}$ by setting $\mathsf{m}_{a_k} = 0$ for $0 \leq k < \ell$.

Online:

1. Set up $\langle b_k \rangle^{\mathbf{B}} = \langle \mathsf{m}_x[k] \rangle^{\mathbf{B}}$ by setting $\mathsf{m}_{b_k} = \mathsf{m}_x[k], \lambda_{b_k}^1 = \lambda_{b_k}^2 = 0$ for $0 \leq k < \ell$.
2. Compute sign $\langle s \rangle^{\mathbf{B}}$ of $\langle x \rangle$ by running a binary PPA circuit on inputs $(\langle a_{\ell-1} \rangle^{\mathbf{B}}, ..., \langle a_0 \rangle^{\mathbf{B}})$ and $(\langle b_{\ell-1} \rangle^{\mathbf{B}}, ..., \langle b_0 \rangle^{\mathbf{B}})$.

Figure 11: ASTRA: Most Significant Bit extraction protocol.

Protocol $\Pi_{\mathsf{MSB}}(\langle x \rangle)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle^{\mathbf{B}}$-shares of $s \in \mathbb{Z}_2$ with $s = 1$ iff $x < 0$.

Setup:

1. $\mathsf{S}_0, \mathsf{S}_1$ sample random $r_k^1 \in \mathbb{Z}_2$ for $0 \leq k < \ell$.
2. Non-interactively set up $\langle b_k \rangle^{\mathbf{B}} = \langle \lambda_x^2[k] \rangle^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\mathsf{m}_{b_k} = \lambda_{b_k}^1 = 0, \lambda_{b_k}^2 = \lambda_x^2[k]$.

Online:

1. S_1 computes and sends $r_k^2 = (\mathsf{m}_x + \lambda_x^1)[k] \oplus r_k^1$ to S_2 for $0 \leq k < \ell$.
2. Non-interactively set up $\langle a_k \rangle^{\mathbf{B}} = \langle (\mathsf{m}_x + \lambda_x^1)[k] \rangle^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\mathsf{m}_{b_k} = r_k^2, \lambda_{b_k}^1 = r_k^1, \lambda_{b_k}^2 = 0$.
3. Compute sign $\langle s \rangle^{\mathbf{B}}$ of $\langle x \rangle$ by running a binary PPA circuit on inputs $(\langle a_{\ell-1} \rangle^{\mathbf{B}}, ..., \langle a_0 \rangle^{\mathbf{B}})$ and $(\langle b_{\ell-1} \rangle^{\mathbf{B}}, ..., \langle b_0 \rangle^{\mathbf{B}})$.

Figure 12: AUXILIATOR: Most Significant Bit extraction protocol.

Protocol $\Pi_{\mathsf{MSB}}(\llbracket x \rrbracket)$

Input: $\llbracket \cdot \rrbracket$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $s \in \mathbb{Z}_2$ with $s = 1$ iff $x < 0$.

Setup:

1. Non-interactively set up $\llbracket a_k \rrbracket^{\mathbf{B}} = \llbracket \lambda_x^0[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\lambda_{a_k}^0 = \lambda_x^0[k], \lambda_{a_k}^1 = \lambda_{a_k}^2 = \mathsf{m}_{a_k} = 0$.

2. Non-interactively set up $\llbracket b_k \rrbracket^{\mathbf{B}} = \llbracket \lambda_x^2[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\lambda_{a_k}^2 = \lambda_x^2[k], \lambda_{a_k}^0 = \lambda_{a_k}^1 = \mathsf{m}_{a_k} = 0$.

3. For $0 \leq k < \ell$, $\mathsf{S}_1, \mathsf{S}_2$ sample random $\lambda_{c_k}^1 \in \mathbb{Z}_2$ and $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ sample random $\lambda_{c_k}^0, \lambda_{c_k}^2 \in \mathbb{Z}_2$.

Online:

1. Non-interactively (partially) set up $\llbracket c_k \rrbracket^{\mathbf{B}} = \llbracket (\mathsf{m}_x + \lambda_x^1)[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by $\mathsf{S}_1, \mathsf{S}_2$ setting $\mathsf{m}_{c_k} = (\mathsf{m}_x + \lambda_x^1)[k] \oplus \lambda_{c_k}^0 \oplus \lambda_{c_k}^1 \oplus \lambda_{c_k}^2$.

2. For $0 \leq k < \ell$, compute $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $s_k, t_k \in \mathbb{Z}_2$ that are the outputs of a full-adder with inputs a_k, b_k, c_k as follows:

   – $\llbracket s_k \rrbracket^{\mathbf{B}} = \llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket b_k \rrbracket^{\mathbf{B}} \oplus \llbracket c_k \rrbracket^{\mathbf{B}}$.

   – $\llbracket t_k \rrbracket^{\mathbf{B}} = \llbracket a_k \rrbracket^{\mathbf{B}} \oplus \big((\llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket b_k \rrbracket^{\mathbf{B}}) \wedge (\llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket c_k \rrbracket^{\mathbf{B}}) \big)$.

3. Compute sign $\llbracket s \rrbracket^{\mathbf{B}}$ of $\llbracket x \rrbracket$ by running a binary PPA circuit on inputs $(\llbracket s_{\ell-1} \rrbracket^{\mathbf{B}}, ..., \llbracket s_0 \rrbracket^{\mathbf{B}})$ and $(\llbracket s_{\ell-2} \rrbracket^{\mathbf{B}}, ..., \llbracket s_0 \rrbracket^{\mathbf{B}}, \llbracket 0 \rrbracket^{\mathbf{B}})$.

Final Verification (batched over all multiplications):

1. S_1 sends m_{c_k} to S_0 for $0 \leq k < \ell$.

2. S_0 now has its shares of $\llbracket c_k \rrbracket^{\mathbf{B}}$ available for further verification.

Figure 13: SOCIUM: Most Significant Bit extraction protocol.

Protocol $\Pi_{\mathsf{MSB}}(\llbracket x \rrbracket)$

Input: $\llbracket \cdot \rrbracket$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $s \in \mathbb{Z}_2$ with $s = 1$ iff $x < 0$.

Setup:

1. Non-interactively set up $\llbracket a_k \rrbracket^{\mathbf{B}} = \llbracket \lambda_x^0[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\lambda_{a_k}^0 = \lambda_x^0[k], \lambda_{a_k}^1 = \lambda_{a_k}^2 = \mathsf{m}_{a_k} = 0$.

2. Non-interactively set up $\llbracket b_k \rrbracket^{\mathbf{B}} = \llbracket \lambda_x^2[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\lambda_{a_k}^2 = \lambda_x^2[k], \lambda_{a_k}^0 = \lambda_{a_k}^1 = \mathsf{m}_{a_k} = 0$.

3. For $0 \leq k < \ell$, $\mathsf{S}_1, \mathsf{S}_2$ sample random $\lambda_{c_k}^1 \in \mathbb{Z}_2$ and $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ sample random $\lambda_{c_k}^0, \lambda_{c_k}^2 \in \mathbb{Z}_2$.

Online:

1. Non-interactively (partially) set up $\llbracket c_k \rrbracket^{\mathbf{B}} = \llbracket (\mathsf{m}_x + \lambda_x^1)[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by $\mathsf{S}_1, \mathsf{S}_2$ setting $\mathsf{m}_{c_k} = (\mathsf{m}_x + \lambda_x^1)[k] \oplus \lambda_{c_k}^0 \oplus \lambda_{c_k}^1 \oplus \lambda_{c_k}^2$.

2. For $0 \leq k < \ell$, compute $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $s_k, t_k \in \mathbb{Z}_2$ that are the outputs of a full-adder with inputs a_k, b_k, c_k as follows:

   – $\llbracket s_k \rrbracket^{\mathbf{B}} = \llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket b_k \rrbracket^{\mathbf{B}} \oplus \llbracket c_k \rrbracket^{\mathbf{B}}$.

   – $\llbracket t_k \rrbracket^{\mathbf{B}} = \llbracket a_k \rrbracket^{\mathbf{B}} \oplus \big((\llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket b_k \rrbracket^{\mathbf{B}}) \wedge (\llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket c_k \rrbracket^{\mathbf{B}}) \big)$.

3. Compute sign $\llbracket s \rrbracket^{\mathbf{B}}$ of $\llbracket x \rrbracket$ by running a binary PPA circuit on inputs $(\llbracket s_{\ell-1} \rrbracket^{\mathbf{B}}, ..., \llbracket s_0 \rrbracket^{\mathbf{B}})$ and $(\llbracket s_{\ell-2} \rrbracket^{\mathbf{B}}, ..., \llbracket s_0 \rrbracket^{\mathbf{B}}, \llbracket 0 \rrbracket^{\mathbf{B}})$.

Final Verification (batched over all multiplications):

1. S_1 sends m_{c_k} and S_2 sends $\mathsf{H}(\mathsf{m}_{c_k})$ to S_0 that then checks consistency for $0 \leq k < \ell$.

2. S_0 now has its shares of $\llbracket c_k \rrbracket^{\mathbf{B}}$ available for further verification.

Figure 14: SWIFT: Most Significant Bit extraction protocol.

1.6. Bit to Arithmetic Conversion

Protocol $\Pi_{\mathsf{BitA}}(\langle x \rangle^{\mathbf{B}})$

Input: $\langle \cdot \rangle^{\mathbf{B}}$-shares of $x \in \mathbb{Z}_2$.

Output: $\langle \cdot \rangle$-shares of $z = (x)^{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$, i.e., z is x padded with $\ell - 1$ 0 bits.

Setup:

1. Generate $[p] = [(\lambda_x^1 + \lambda_x^2)^{\mathsf{a}}]$:

   – $\mathsf{S}_0, \mathsf{S}_1$ sample random $p^1 \in \mathbb{Z}_{2^\ell}$.

   – S_0 computes and sends $p^2 = (\lambda_x^1 + \lambda_x^2)^{\mathsf{a}} - p^1$ to S_2.

2. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following:

   – S_1: $q^1 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot p^1 - \lambda_z^1$.

   – S_2: $q^2 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot p^2 - \lambda_z^2$.

2. S_1 sends q^1 to S_2 and S_2 sends q^2 to S_1.

3. Compute $\mathsf{m}_z = q^1 + q^2 + (\mathsf{m}_x)^{\mathsf{a}}$.

Figure 15: ASTRA: Bit to arithmetic conversion.

Protocol $\Pi_{\mathsf{BitA}}(\langle x \rangle^{\mathbf{B}})$

Input: $\langle \cdot \rangle^{\mathbf{B}}$-shares of $x \in \mathbb{Z}_2$.

Output: $\langle \cdot \rangle$-shares of $z = (x)^{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$, i.e., z is x padded with $\ell - 1$ 0 bits.

Setup:

1. Non-interactively set up $[p] = [(\lambda_x^1)^{\mathsf{a}}]$ by setting $p^1 = (\lambda_x^1)^{\mathsf{a}}, p^2 = 0$.

2. Non-interactively set up $[q] = [(\lambda_x^2)^{\mathsf{a}}]$ by setting $p^1 = 0, p^2 = (\lambda_x^2)^{\mathsf{a}}$.

3. Invoke $\mathcal{F}_{\mathsf{MultPre}}([p], [q])$ to obtain $[pq]$.

4. Compute $[r] = [(\lambda_x^1 \oplus \lambda_x^2)^{\mathsf{a}}] = [p] + [q] - 2[pq]$.

5. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following:

   – S_1: $s^1 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot r^1 - \lambda_z^1$.

   – S_2: $s^2 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot r^2 - \lambda_z^2$.

2. S_1 sends s^1 to S_2 and S_2 sends s^2 to S_1.

3. Compute $\mathsf{m}_z = s^1 + s^2 + (\mathsf{m}_x)^{\mathsf{a}}$.

Figure 16: AUXILIATOR: Bit to arithmetic conversion.

Protocol $\Pi_{\mathsf{BitA}}(\llbracket x \rrbracket^{\mathbf{B}})$

Input: $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $x \in \mathbb{Z}_2$.

Output: $\llbracket \cdot \rrbracket$-shares of $z = (x)^{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$, i.e., z is x padded with $\ell - 1$ 0 bits.

Setup:

1. Non-interactively set up $\langle p \rangle = \langle (\lambda_x^0)^{\mathsf{a}} \rangle$ by setting $p^0 = (\lambda_x^0)^{\mathsf{a}}, p^1 = p^2 = 0$.

2. Non-interactively set up $\langle q \rangle = \langle (\lambda_x^1)^{\mathsf{a}} \rangle$ by setting $q^1 = (\lambda_x^1)^{\mathsf{a}}, q^0 = q^2 = 0$.

3. Non-interactively set up $\langle r \rangle = \langle (\lambda_x^0)^{\mathsf{a}} \rangle$ by setting $r^2 = (\lambda_x^2)^{\mathsf{a}}, r^0 = r^1 = 0$.

4. Invoke $\mathcal{F}_{\mathsf{MultPre}}(\langle p \rangle, \langle q \rangle)$ to obtain $\langle pq \rangle$. **Note that SWIFT func-**

tionality $\mathcal{F}_{\mathsf{MultPre}}$ is used instead of SOCIUM functionality $\mathcal{F}'_{\mathsf{MultPre}}$ here to obtain a full replicated sharing. Still, the simplified verification of SOCIUM can be used in its instantiation.

5. Compute $\langle s\rangle = \langle(\lambda_x^0 \oplus \lambda_x^1)^{\mathsf{a}}\rangle = \langle p\rangle + \langle q\rangle - 2\langle pq\rangle$.

6. Invoke $\mathcal{F}'_{\mathsf{MultPre}}(\langle s\rangle, \langle c\rangle)$ to obtain $\langle sc\rangle'$.

7. Compute $\langle t\rangle' = \langle(\lambda_x^0 \oplus \lambda_x^1 \oplus \lambda_x^2)^{\mathsf{a}}\rangle' = \langle s\rangle' + \langle c\rangle' - 2\langle sc\rangle'$.

8. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following (S_0 delays the step until the final verification):

- S_1 : $u^0 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^0 - \lambda_z^0$
- S_1 : $u^1 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^1 - \lambda_z^1$
- $\mathsf{S}_2, \mathsf{S}_0$: $u^2 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^2 - \lambda_z^2$

2. S_2 sends u^2 to S_1 and S_1 sends $u^0 + u^1$ to S_2.

3. $\mathsf{S}_1, \mathsf{S}_2$ set $\mathsf{m}_z = (\mathsf{m}_x)^{\mathsf{a}} + u^0 + u^1 + u^2$.

Final Verification (batched over all multiplications):

1. S_1 sends $u^0 + u^1$ to S_0.

2. S_0 computes u^2 and reconstructs $\mathsf{m}_z = (\mathsf{m}_x)^{\mathsf{a}} + u^0 + u^1 + u^2$.

3. S_0 sends $\mathcal{H}(u^2)$ to S_1 that then checks consistency to the u^2 it received from S_2.

Figure 17: SOCIUM: Bit to arithmetic conversion.

Protocol $\Pi_{\mathsf{BitA}}(\llbracket x \rrbracket^{\mathbf{B}})$

Input: $\llbracket\cdot\rrbracket^{\mathbf{B}}$-shares of $x \in \mathbb{Z}_2$.

Output: $\llbracket\cdot\rrbracket$-shares of $z = (x)^{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$, i.e., z is x padded with $\ell - 1$ 0 bits.

Setup:

1. Non-interactively set up $\langle p\rangle = \langle(\lambda_x^0)^{\mathsf{a}}\rangle$ by setting $p^0 = (\lambda_x^0)^{\mathsf{a}}, p^1 = p^2 = 0$.

2. Non-interactively set up $\langle q\rangle = \langle(\lambda_x^1)^{\mathsf{a}}\rangle$ by setting $q^1 = (\lambda_x^1)^{\mathsf{a}}, q^0 = q^2 = 0$.

3. Non-interactively set up $\langle r\rangle = \langle(\lambda_x^0)^{\mathsf{a}}\rangle$ by setting $r^2 = (\lambda_x^2)^{\mathsf{a}}, r^0 = r^1 = 0$.

4. Invoke $\mathcal{F}_{\mathsf{MultPre}}(\langle p\rangle, \langle q\rangle)$ to obtain $\langle pq\rangle$.

5. Compute $\langle s\rangle = \langle(\lambda_x^0 \oplus \lambda_x^1)^{\mathsf{a}}\rangle = \langle p\rangle + \langle q\rangle - 2\langle pq\rangle$.

6. Invoke $\mathcal{F}_{\mathsf{MultPre}}(\langle s\rangle, \langle c\rangle)$ to obtain $\langle sc\rangle$.

7. Compute $\langle t\rangle = \langle(\lambda_x^0 \oplus \lambda_x^1 \oplus \lambda_x^2)^{\mathsf{a}}\rangle = \langle s\rangle + \langle c\rangle - 2\langle sc\rangle$.

8. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following (S_0 delays the step until the final verification):

- $\mathsf{S}_0, \mathsf{S}_1$: $u^0 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^0 - \lambda_z^0$
- $\mathsf{S}_1, \mathsf{S}_2$: $u^1 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^1 - \lambda_z^1$
- $\mathsf{S}_2, \mathsf{S}_0$: $u^2 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^2 - \lambda_z^2$

2. S_2 sends u^2 to S_1 and S_1 sends u^0 to S_2.

3. $\mathsf{S}_1, \mathsf{S}_2$ set $\mathsf{m}_z = (\mathsf{m}_x)^{\mathsf{a}} + u^0 + u^1 + u^2$.

Final Verification (batched over all multiplications):

1. S_1 sends u^1 and S_2 sends $\mathsf{H}(u^1)$ to S_0 that then checks for consistency.

2. S_0 computes u^0, u^2 and reconstructs $\mathsf{m}_z = (\mathsf{m}_x)^{\mathsf{a}} + u^0 + u^1 + u^2$.

3. S_0 sends $\mathcal{H}(u^2)$ to S_1 and $\mathcal{H}(u^0)$ to S_2 that then check consistency to the values received in online step 2.

Figure 18: SWIFT: Bit to arithmetic conversion.

1.7. Binary to Arithmetic Conversion

For our comparison to ELSA [3] we require a conversion of an integer decomposed into its bits into an arithmetic sharing. This is only needed for AUXILIATOR.

Protocol $\Pi_{\mathsf{B2A}}(\langle x_0\rangle^{\mathbf{B}}, ..., \langle x_{k-1}\rangle^{\mathbf{B}})$

Input: $\langle\cdot\rangle^{\mathbf{B}}$-shares of $x_{k-1}, ..., x_0 \in \mathbb{Z}_2$ for some $k \in \mathbb{N}$.

Output: $\langle\cdot\rangle$-shares of $z \in \mathbb{Z}_{2^\ell}$ where z as binary encoding corresponds to $x_{k-1}, ..., x_0$, i.e., $z = \sum_{i=0}^{k-1} 2^i x_i$.

Setup:

1. Non-interactively set up $[p_i] = [(\lambda_{x_i}^1)^{\mathsf{a}}]$ by setting $p_i^1 = (\lambda_{x_i}^1)^{\mathsf{a}}, p_i^2 = 0$ for $0 \le i < k$.

2. Non-interactively set up $[q_i] = [(\lambda_{x_i}^2)^{\mathsf{a}}]$ by setting $p_i^1 = 0, p_i^2 = (\lambda_{x_i}^2)^{\mathsf{a}}$ for $0 \le i < k$.

3. Invoke $\mathcal{F}_{\mathsf{MultPre}}([p_i], [q_i])$ to obtain $[p_i q_i]$ for $0 \le i < k$.

4. Compute $[r_i] = [(\lambda_{x_i}^1 \oplus \lambda_{x_i}^2)^{\mathsf{a}}] = [p_i] + [q_i] - 2[p_i q_i]$ for $0 \le i < k$.

5. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following:

- S_1: $s^1 = \sum_{i=0}^{k-1} \left(2^i \cdot (1 - 2(\mathsf{m}_{x_i})^{\mathsf{a}}) \cdot r_i^1\right) - \lambda_z^1$.
- S_2: $s^2 = \sum_{i=0}^{k-1} \left(2^i \cdot (1 - 2(\mathsf{m}_{x_i})^{\mathsf{a}}) \cdot r_i^2\right) - \lambda_z^2$.

2. S_1 sends s^1 to S_2 and S_2 sends s^2 to S_1.

3. Compute $\mathsf{m}_z = s^1 + s^2 + \sum_{i=0}^{k-1} \left(2^i \cdot (\mathsf{m}_{x_i})^{\mathsf{a}}\right)$.

Figure 19: AUXILIATOR: Binary to arithmetic conversion.

References

[1] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction," in *CCSW@CCS*, 2019.

[2] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning," in *USENIX Security*, 2021.

[3] M. Rathee, C. Shen, S. Wagh, and R. A. Popa, "ELSA: Secure Aggregation for Federated Learning with Malicious Actors," in *IEEE S&P*, 2023.

[4] A. Suresh, "MPCLeague: Robust MPC Platform for Privacy-Preserving Machine Learning," *PhD Thesis*, vol. abs/2112.13338, 2021, https://arxiv.org/abs/2112.13338.