Force: Making 4PC $>$ 4 $\times$ PC in Privacy Preserving Machine Learning on GPU

Tianxiang Dai*, Li Duan*†, Yufan Jiang*‡✉, Yong Li*✉, Fei Mei*§, Yulian Sun*¶
* *Huawei European Research Center, Germany*
† *Paderborn University, Germany*
‡ *Karlsruhe Institute of Technology, Germany*
§ *Technical University of Munich, Germany*
¶ *Technical University of Darmstadt, Germany*
yufan.jiang@{huawei.com, partner.kit.edu}
{tianxiangdai, li.duan, yong.li1, fei.mei, yulian.sun1}@huawei.com

***Abstract*—Tremendous efforts have been made to improve the efficiency of secure Multi-Party Computation (MPC), which allows $n \geq 2$ parties to jointly evaluate a target function without leaking their own private inputs. It has been confirmed by previous researchers that 3-Party Computation (3PC) and outsourcing computations to GPUs can lead to huge performance improvement of MPC in computationally intensive tasks such as Privacy-Preserving Machine Learning (PPML). A natural question to ask is whether super-linear performance gain is possible for a linear increase in resources. In this paper, we give an affirmative answer to this question.**

We propose Force, an extremely efficient 4PC system for PPML. To the best of our knowledge, each party in Force enjoys the *least number of local computations*, *smallest graphic memory consumption* and *lowest data exchanges* between parties. This is achieved by introducing a new sharing type $\mathcal{X}$-share along with MPC protocols in privacy-preserving training and inference that are *semi-honest* secure with an *honest-majority*. Our contribution does not stop at theory. We also propose engineering optimizations and verify the high performance of the protocols with implementation and experiments. By comparing the results with state-of-the-art researches, we showcase that Force is sound and extremely efficient, as it can improve the PPML performance by a factor of 2 to 38 compared with other latest GPU-based *semi-honest* secure systems, such as Piranha (including SecureML, Falcon and FantasticFour), CryptGPU and CrypTen.

***Index Terms*—MPC, privacy-preserving machine learning, four party computation**

1. Introduction

Values have been constantly generated from machine learning (ML) over mass data collected from different users. On the other hand, the importance of privacy and data security have also been increasingly recognized, and more than 70% of world countries and regions have installed legislation for privacy and data security [69]. Technically, it is a good starting point to always keep sensitive data at local storage and never reveal them on the Internet in plaintext, but to preserve the usability of data distributed across owners remains challenging.

Secure multi-party computation protocols (MPC) have been designed for multiple parties to jointly compute a function without revealing their own secret inputs. Starting from the two-party case (2PC), frameworks have been proposed [38, 41, 59] to offer various trade-offs of security and performance. Extending 2PC to three party protocols (3PC), especially by tailoring the secret shares such that a single corrupted party cannot learn anything useful about the complete secret value, leads to a large leap in performance [6, 7] in the honest majority setting. Follow-up works [1, 42, 67] have proposed algorithmic and engineering optimizations to bring MPC closer to real-world and *high-throughput* applications, such as privacy-preserving machine learning (PPML), although a giant gap still remains.

Thus natural questions to ask are: *Can 4PC be non-trivially faster than 3PC? Can 4PC undertake tasks that are previously impossible in 3PC?* In this paper, we give an affirmative answer to these questions. By introducing a new sharing type $\mathcal{X}$-share and a new set of protocols, our new Force framework for 4PC is not only secure at its cryptographic core, but more importantly, it outperforms cutting-edge 2PC/3PC/4PC solutions for privacy-preserving machine learning remarkably by a factor of 2 to 38.

More specifically, we make the following contributions.

- **New 4PC Protocols for Multiplication on GPU**. Our idea is inspired by the 3PC multiplication based on replicated sharing [1, 13], but with a brand new sharing types called $\mathcal{X}$-share (§4.1.3) tailored for four non-colluding parties. Benefiting from these new shares, our 4PC matrix multiplication protocol achieves the lowest number of local multiplications and number of ring elements sent/received by each party (See Table 2).
- **New 4PC Protocol for Comparison, Sharing Type Conversion and Truncation.** There are a large number of non-linear operations in PPML, so we still follow the mixed-protocol approach proposed by ABY-style MPC frameworks [20, 54, 59] and design novel share conversion protocols. With a new type of correlated

TABLE 1: Comparison of Force and state-of-the-art works against semi-honest adversaries. In both LAN and WAN setting, we show our max boosting factor against other frameworks. We denote Parallel-prefix Adders circuit as PPA.

Setting	Ref.	Year	Platform	Linear Ops	Non-Linear Ops	LAN		WAN
						Training	Inference	Inference
2PC	Cheetah[34]	2022	**CPU**	FHE	cOT	-	**1234x**	**70x**
	P-SecureML[55, 73]	2022	GPU/CPU	2-out-of-2 Arithmetic	PPA with 2o2/GC	**14x**	**5.8x**	**13x**
3PC	CryptGPU[67]	2021	GPU	Replicated Sharing (RS)	PPA with RS	**6.5x**	**14x**	**10x**
	P-Falcon[71, 73]	2022	GPU/CPU	RS	PPA with RS	**2.1x**	**3.1x**	**2.4x**
4PC	CrypTen[42]	2021	GPU	4-out-of-4 Arithmetic	PPA with 4o4	**38x**	**10x**	**29x**
	P-FantasticFour[17, 73]	2021	GPU/CPU	RS	PPA with RS	**4.7x**	**7.4x**	**10x**
	Force	2022	GPU	$\mathcal{X}$-share	PPA with $\mathcal{X}$-share	**1**	**1**	**1**

TABLE 2: Comparison of Force and existing works in multiplication at each party, secure against **semi-honest** adversary. ℓ: bit-length of a ring element, κ: the statistical security parameter, Comm: bit-length of data communicated, Mult: number of local multiplication, Add: number of local addition.

Setting	Ref.	Setup	Online		Local	
		Comm	Comm	Rounds	Mult	Add
2PC	P-SecureML[55]	$2\ell(\kappa+\ell)$	4ℓ	1	3	5
3PC	CryptGPU[67]	0	2ℓ	1	3	3
	P-Falcon[73]	0	2ℓ	1	3	3
4PC	CrypTen[42]	$2\ell(\kappa+\ell)$	8ℓ	1	3	9
	P-FantasticFour[73][17]	0	4ℓ	1	7	5
	Force	**0**	$\mathbf{2\ell}$	**1**	**1**	**2**

randomness, the $\mathcal{X}$-dabit (§4.3.1), our A2B, B2A and comparison protocol also achieve the least computational and communication costs. In addition, thanks to the symmetry of participants in 4PC, we can eliminate the communication in 3PC by turning to the communication-free truncation proposed by SecureML [55] to keep the precision consistent.

- **Overall Optimized Performance in PPML.** We also make fair comparison between the protocols with the same security parameters, precision, the newest implementation and hardware. For a better insight, we also include the latest CPU-only framework Cheetah [34] for inference comparison. An overview of the evaluation is shown in Table 1.
- **Optimized Graphic Memory Usage.** Graphic memory consumption limits the performance and sometimes even prohibits executing PPML, especially training. On the other hand, unlike [38, 49, 53, 62], which provide inference-only implementations, our aim is to train real-world ML models such as `VGG16` [66] with large batch size in MPC over GPU.
  Given $\mathcal{X}$-share (§4.1.3) and our improved carry out implementation (§5.3), Force greatly reduce the graphic memory consumption of each party so that it can perform PPML training of one large dataset, `ImageNet` [64] on large networks like `VGG16` with BatchSize $=$ 16, which was not possible in prior solutions.

The paper is organized as follows. After the introduction, we survey related work in §2. Notations, operations in PPML and other primitives are introduced in §3. Our new sharing type and 4PC protocols are presented in §4. The implementation and evaluation results in PPML are presented in §5 and §6. Finally, the conclusion and open questions can be found in §7.

2. Related Work

The related work is summarized concisely here, and more details of the primitives, such as correlated randomness, detailed sharing types and protocols, can be found in §3 and §4.

2.1. Choice of Sharing Types

Intuitively, shares are values distributed among parties that can be used to reconstruct the original inputs.

Given a target function, types of shares determine the efficiency of protocols. If two parties hold boolean shares (x_0, x_1) of $x \in \{0,1\}^\ell$, i.e., $\mathbf{P_A}$ holds x_0 and $\mathbf{P_B}$ has x_1 with $x_0 \oplus x_1 = x$, then efficient 2PC protocols for evaluating binary circuits consisting of AND and XOR gates can be executed [5, 74]. But boolean shares cannot reach high throughput in arithmetic operations, as the number of gates is too large even for moderate sized inputs. For example, a 64-bit multiplier in Bristol fashion circuit provided by SCALE-MAMBA [47] needs 4033 AND-gates with a depth of 63, so evaluating such circuits with BGW protocol [5] or GMW [27] would result in an explosion in data volume exchanged. Arithmetic shares with $x_0 + x_1 = x \in \mathbb{Z}_M$ are more appropriate for algebraic multiplication protocols. This sharing type enables high-throughput multiplication protocols in the pre-processing model, where correlated randomness such as Beaver's multiplication triples [4] has been generated in the set-up phase [41] to accelerate the online phase. The impact of sharing types thus motivates the **mixed protocol** approach adopted by ABY (2015) [20] in computing complicated functions in 2PC for good overall performance. More specifically, Arithmetic shares are used for integer addition and multiplication, while Boolean shares and Yao shares are used for non-linear operation such as comparison, and sharing type conversion protocols (A2B,A2Y,Y2B) are executed when necessary. The major update in ABY2.0 also lies in its new $\langle\rangle$-share (see Section 3.1.3 in [59]), which reduces the the communication during multiplication by half.

Going beyond 2PC, threshold sharing types have often been proposed to reduce local computation and communicated data volumes. To outperform generic BGW [5],

Sharemind [6, 7] carefully chose $(2,3)$-shares for 3PC. Replicated shares, a special kind of $(2,3)$-share proposed by Catrina *et al.* in 2010 [13], are extended by Araki *et al.* in 2016 [1] and used by CryptGPU [67] in 2021 to replace Beaver triples with zero-shares during multiplication, giving further performance improvement. Therefore, it is logical to consider new sharing types for efficient 4PC or beyond.

2.2. Privacy Preserving Machine Learning

One of the major challenges for privacy-preserving machine learning (PPML) is *high-throughput*. As justified in [20, 41, 59], correlated randomness, optimal sharing types and mixed protocol with pre-processing are extremely helpful in achieving overall high-throughput. This *de facto* standard has been followed by researchers from early attempts of PPML till now.

To the best of our knowledge, executing neural networks (NN) and linear/logic regression in 2PC was first attempted by SecureML [55] in 2017. The cryptographic core of SecureML is consisted of ABY shares with correlated randomness, a suite of 2PC protocols for linear/non-linear operation and an efficient 2PC truncation, which helps avoid overflow with high precision. Later attempts like miniONN (2017) [52], secureNN (2019) [70], Falcon (2020) [71], Cheetah (2022) [34] and [32, 38, 49, 53, 54, 62, 75] still follow the mixed protocol approach with various optimization for multiplication and approximation methods for other non-linear operation, such as ReLU and Sigmoid activation functions. These attempts mainly focus on demonstrating the asymptotic feasibility of running PPML with 2/3PC or homomorphic encryption (HE) and provable security of the system. This might be the primary reason why few of them has taken advantages of Graphical Processing Unit (GPU) or adapt the solutions for specific ML frameworks.

2.3. PPML on GPU

Researches on implementing PPML on GPU can be seen as a tour that starts from two ends and finally meets in the middle.

The work of Pu *et al.* [61] in 2011 might be the first implementation of Yao's Garbled Circuit (GC) [23] on GPU. The GFLOPS[1] per dollar and watt, as well as optimized parallelism of GPU, motivated Husted *et al.* [35] and Frederiksen and Nielsen [24] to work on more modern protocols for GC in 2013. Later, cuHE [16] investigated the accessibility of homomorphic encryption (HE) on GPU. These pioneering works uncovered the potential of GPU-friendly MPC, which can usually improve the performance by a factor up to 60 [24] compared with the CPU-based peers in high-throughput scenarios.

On the other hand, privacy and security features are being constantly considered and adopted in ML frameworks. Secure aggregation and Federated Learning (FL) [8] were proposed by Google in 2016 for training shared models over data distributed across users. TensorFlow Privacy was officially announced [28] in 2019, which incorporates differential privacy (DP) [21] during the training process. Although being quite efficient, FL and DP cannot guarantee the same security as MPC does [39, 68].

Finally, the two ends meet at CrypTen (2020) [42]. While CrypTen still maintains an ABY-style cryptographic core, the underlying MPC protocols in CrypTen are abstracted in a more ML-oriented way so that it can offer PyTorch-like [58] interfaces for ML practitioners, making the PPML framework more approachable for non-cryptographers and extensible for arbitrary number of parties.

CryptGPU [67] chooses to extend CrypTen with other GPU-friendly MPC components in a special case: 3PC. As mentioned before, the most important measure to boost MPC performance in CryptGPU are $(2,3)$-shares and the corresponding multiplication/AND protocols besides engineering optimizations. Watson *et al.* propose Piranha [73], a modular framework for accelerating generic secret sharing-based MPC protocols over GPU. With novel engineering optimizations, Piranha can train real PPML model such as VGG [66], which was previously impossible on CryptGPU.

For other approaches to implement PPML such as using designated hardware, we refer the reader to the nice surveys [10, 30, 57].

3. Preliminaries

TABLE 3: Notations.

Term	Meaning
$\mathbf{P_i}$	party i
ℓ	length of an arithmetic value
ℓ^{B}	length of a binary value
n	number of parties
p	number of bits in decimal part
r	randomness
θ, ψ, ϕ	share-mode of a secret value x, where $\theta, \psi, \phi \in \{\mathsf{AC}, \mathsf{AB}\}$
$[x]_{\mathsf{RS}}$	a value x is arithmetically replicated shared over $\mathbb{Z}_{2^\ell}$ among three parties
$[x]_{\mathsf{AC}}$	a value x is arithmetically AC shared over $\mathbb{Z}_{2^\ell}$, where $\mathbf{P_A}$, $\mathbf{P_B}$ hold the same local share, as well as $\mathbf{P_C}$, $\mathbf{P_D}$
$[x]_{\mathsf{AB}}$	a value x is arithmetically AB shared over $\mathbb{Z}_{2^\ell}$, where $\mathbf{P_A}$, $\mathbf{P_C}$ hold the same local share, as well as $\mathbf{P_B}$, $\mathbf{P_D}$
$[x]_{\mathsf{4o4}}$	a value x is arithmetically 4-out-of-4 shared over $\mathbb{Z}_{2^\ell}$
$[x]_{\psi}^{\mathbf{P_i}}$	the local share of $\mathbf{P_i}$ of shared value x
$\langle x \rangle_\theta$	a value x is boolean θ shared
$\langle x_i \rangle_\theta$	the ith bit of x is boolean shared in θ-mode over $\mathbb{Z}_2$
$[\![r]\!]_{\psi \mathsf{to} \phi}$	CMS for change share-mode protocol
$r^{\mathbf{P_i}}$	$\mathbf{P_i}$'s local share of a *zero sharing*
Π_f	a protocol for computing function f
$\mathcal{F}_{\mathsf{4PC}}^f$	ideal functionality for function f in 4PC
sid	session id
$a \xleftarrow{\$} S$	sample a from S uniformly at random

We summarize all important notations in Table 3, and others will be explained when necessary.

1. Giga (10^9) Floating Point Operation Per Second

3.1. Layer and Operations in CNN

The development in deep learning and convolutional neural network (CNN) has brought revolution to computer vision and other fields [31, 46, 66]. A CNN is usually consisted of stacked convolutional layers combined with activation, (optional) normalization layers, and pooling layers, and followed by fully connected layers. Here we briefly introduce each layer.

Convolutional layer A convolutional layer is composed of d convolutional kernels (See Figure 16 in Appendix). The input and kernel have same channel depth but different height and width After one kernel slides over the whole image, a new feature map is produced, i.e., d kernels produce d feature maps. Stacked d feature maps construct the output of convolution layer. Convolutional layer mainly involves matrix multiplication.

Activation layer Activation layer introduces nonlinearity to the neural network. Although typical activation functions include ReLU: $f(x) = \max(0, x)$, Sigmoid : $f(x) = \frac{1}{1+\exp^{-x}}$, and Tanh : $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, in most CNN, ReLU is preferred, which needs comparison with zero.

Pooling layer Common pooling methods include max pooling and average pooling. Max pooling selects the largest value from selected pooling region, while average pooling computes the average value of the region, i.e., division by a public constant is included.

Full Connected Layer FC layer computation mainly involves matrix multiplication and addition. Let $\mathbf{W}$ be the $m \times n$ weight matrix. Given the input vector $\mathbf{x} = [x_0, x_1, x_2, \ldots, x_n]^T$, the bias vector $\mathbf{b} = [b_0, b_1, b_2, \ldots, b_m]^T$, the output vector $\mathbf{y}$ is computed as $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$.

Optional Normalization Layer Batch normalization(BN) [37] is a typical optimization algorithm in neural network. For a batch input of size (m, d, h, w), each sample has d feature maps. Firstly, normalization is done over each feature maps through all the samples in the batch as $\hat{x}_i^j = \frac{x_i^j - E(x^j)}{\sqrt{Var(x^j)}}$ and $E(x^j) = \frac{1}{m}\sum_{k=1}^{k=m} x_k^j$. Then BN result is achieved by scale and shift as $y_i = BN_{\gamma,\beta}(x_i)$ in which $y_i^j = \gamma\hat{x}_i^j + \beta$. The scale γ and shift β are trainable parameters.

Training and Inference of Neural Network The network training includes forward propagation and back propagation. In the forward propagation, the computation is performed sequentially following the network structure from the bottom layer until the top output layer.

When training, the loss is computed after each forward propagation to evaluate the distance between prediction and true value. In the back propagation, optimization of the network parameters is done to minimize loss. Stochastic Gradient Descent (SGD) is the optimization algorithm we implement, where multiplication and division-by-constant are used.

3.2. From Layers to Protocols in Force

It is straightforward to see that each layer in CNN can be decomposed into more fundamental operations, such as multiplication, division-by-constant and comparison. When the MPC protocols for each operation are improved in efficiency, so will the whole system. Therefore, we resort to the modular approach to organize the building blocks of Force as shown in Figure 1. where operations and protocols on each level depends on one or more components below, which are also protocols.

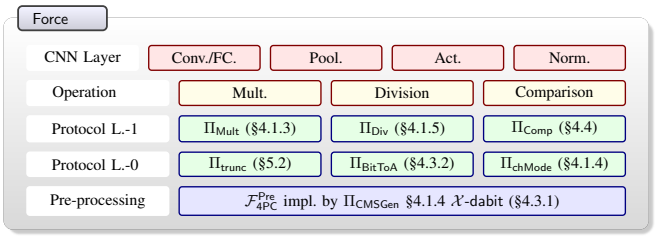


Fig. 1: Overview of Force protocols

From bottom up, correlated randomness is prepared during the pre-processing phase by $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$ for all parties. When executing protocols above, parties can use the correlated randomness and their private inputs to compute the operation and finally assemble the results.

3.3. Correlated Randomness

Correlated randomness are random values with special (algebraic) structural relations that are generated during the pre-processing phase [20, 41, 54, 59] to accelerate the online phase in MPC.

3.3.1. Replicated Shared Secrets and Zero Shares. As defined in [1], a secret value $x \in \mathbb{Z}_{2^\ell}$ is said to be *replicated shared* in 3PC, if three random values $x_0, x_1, x_2 \in \mathbb{Z}_{2^\ell}$ are sampled with $x = x_0 + x_1 + x_2$, and the pairs $(x_0, x_1), (x_1, x_2)$ and (x_2, x_0) are owned by each of the three parties respectively. We denote such sharing type as $[\,]_{\mathsf{RS}}$. Addition and subtraction of two replicated shares $[x]_{\mathsf{RS}}$ and $[y]_{\mathsf{RS}}$ can be locally computed by parties.

The multiplication of $[x]_{\mathsf{RS}}$ and $[y]_{\mathsf{RS}}$ in 3PC, however, requires parties to interact. More specifically, $\mathbf{P_i}$ is able to compute $z_i = x_i y_i + x_{i+1} y_i + x_i y_{i+1}$, yielding a 3-out-of-3 sharing of xy. In order to recover the replicated share $[xy]_{\mathsf{RS}}$, $\mathbf{P_i}$ has to *re-share* their masked local result $z_i + \alpha_i$ to one of the other two parties, where $\sum \alpha_i = 0$. Such *zero sharing* is the correlated randomness that can be derived from a pseudorandom function (PRF) $\mathsf{PRF}()$ with shared keys [67].

We also call such a sharing type as replicated share in general, if any share value x_i is held by more than one party. For better readability, we defer the definition of our new replicated share $\mathcal{X}$-share to §4.1, together with the corresponding multiplication protocol in 4PC.

3.3.2. For Type Conversion : dabit**.** The dabit (doubly authenticated bit) is a type of correlated randomness proposed by Rotaru and Wood [63] to mainly support secure comparison protocol and sharing type conversion. Let $b \xleftarrow{\$} \{0,1\}$ be the randomness to be shared, $\sum$ the arithmetic sum in the ring, $\oplus$ the binary XOR operation. Formally, dabit is defined as

$$\mathsf{dabit} := ([b], \langle b \rangle) \text{ such that } b = \sum [b], b = \oplus \langle b \rangle.$$

To securely generate dabits [63] uses the same techniques explained in [56] such as cut-and-choose in the preprocessing phase.

3.3.3. Extended dabit **:** edabit**.** Recent work [22] of Escudero *et al.* extends dabit to edabit (extended doubly authenticated bit). Similar to dabit, an edabit is a tuple of shares for $b = (b_0, \cdots, b_{\ell-1}) \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ defined as

$$\mathsf{edabit} := ([b],\ \langle b \rangle := (\langle b_0 \rangle, \langle b_1 \rangle, \cdots, \langle b_{l-1} \rangle))$$
$$\text{with } b = \sum [b] \text{ and } b_i = \oplus \langle b_i \rangle.$$

Note that parties can generate edabit by simply combining ℓ dabit, which can be performed locally. Meanwhile, it is also possible to directly generate edabit without generating dabit in advance. [22] proposes a protocol to convert locally generated edabits to a global effective edabit by evaluating n-input binary adder and then executing a Boolean-to-Arithmetic (B2A) protocol.

3.4. Threat Model

A semi-honest adversary cannot deviate from the protocol description, but may try to infer information about the secret input. As a well studied model, security against semi-honest adversaries [50] in the honest majority setting often leads to 2PC and 3PC protocols with good efficiency [1, 42, 54, 55, 59, 62, 67, 75], while the ones with *malicious* security [2, 14, 19, 25, 40, 41, 51, 71, 72] are still too heavy for large-scale applications in practice [23]. The **honest majority setting** is also adopted by 4PC frameworks with semi-honest or malicious security [9, 15, 17, 42, 43, 44, 60], where (strictly) less than one half of the parties can be controlled by an adversary.

We assume confidential, authenticated, and peer-to-peer channels between different parties. Thanks to the channel, the adversary can only read, delay or delete messages as any non-trivial modification can be detected by the parties.

An n-party protocol can be seen as a (probabilistic) process that maps n inputs to n outputs (one for each participant). We name such a process as a **functionality**. We denote a n-party input as

$$f = (f_1, f_2, \cdots, f_n) : \left(\{0,1\}^*\right)^n \rightarrow \left(\{0,1\}^*\right)^n, \qquad (1)$$

where $f_1(), \cdots, f_n()$ are process executed by party 1 to party n, respectively. If we refer to a crytographically secure functionality, we also call it an **ideal functionality** or **ideal** in short.

Let $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$ denote the output of an environment machine $\mathcal{Z}$ interacting with the adversary $\mathcal{A}$ executing the protocol Π in the real world. Let $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the output of $\mathcal{Z}$ interacting with a simulator $\mathcal{S}$ connected to an ideal functionality $\mathcal{F}$ in the ideal world.

Definition 1 (UC security)**.** *Let $\mathcal{F}$ be a four-party functionality and let Π be a four-party protocol that computes $\mathcal{F}$. Protocol Π is said to* ***uc-realizes $\mathcal{F}$ with abort in the presence of static semi-honest adversaries*** *if for every non-uniform probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists a non-uniform PPT adversary $\mathcal{S}$, such that for any environment $\mathcal{Z}$*

$$\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \stackrel{c}{\equiv} \mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}.$$

We follow the universally composable framework (UC) described in detail in [11]. More specifically, we use the hybrid model, where provably UC-secure components are abstracted as an ideals in the next proof, to analyze the security of all components layer by layer in the honest majority setting.

4. 4PC Protocols

We construct efficient 4PC protocols as building blocks of Force for PPML. In §4.1, we introduce our new sharing type $\mathcal{X}$-share and how parties can perform 4PC fixed-point computations. We highlight that the multiplication based on $\mathcal{X}$-share reduces the local computation of each party to only one multiplication. To best of our knowledge, this becomes the least computation cost comparing to other sharing construction such as replicated sharing or 2-out-of-2 sharing. In §4.1.4 and §4.3 we show how to utilize a $\mathcal{X}$-dabit transmitted from dabit [63] to perform conversions between sharing types and share-mode.

In all the protocol descriptions, we use the term public parameters to denote all security parameters and ciphersuites identifiers, and sid the session identifier.

4.1. $\mathcal{X}$-share and Arithmetic Computation

In this section, we show how to perform computations with decimal fixed-point arithmetic shares with $\mathcal{X}$-share. We define a fixed point value as a ℓ bit integer using two's complement representation, consists of both integer part and decimal part with $\ell - p$ bits and p bits respectively. Normally, addition and subtraction will be directly performed on $\mathbb{Z}_{2^\ell}$, since the result is supposed to remain below 2^ℓ. Meanwhile, although the multiplication could be performed in the same manner, the result must be divided by 2^p to maintain the same p decimal bit precision.

4.1.1. $\mathcal{X}$**-**share **and Share-mode.** We begin by introducing our new sharing type $\mathcal{X}$-share used in our 4PC computations. $\mathcal{X}$-share can work over both $\mathbb{Z}_{2^\ell}$ and $\mathbb{Z}_2$ rings in two modes.

- $[\cdot]_{\mathsf{AC}}$-sharing : We say that a value x is $[\cdot]_{\mathsf{AC}}$-shared among parties $\{\mathbf{P_i}\}$, if $\mathbf{P_A}$ and $\mathbf{P_B}$ hold the same

Share	$[x]_{\mathsf{AC}}$	$[y]_{\mathsf{AB}}$	$[xy]_{\mathsf{4o4}}$	Resharing as $[xy]_{\mathsf{AC}}$
$\mathbf{P_A}$	x_0	y_0	x_0y_0	$x_0y_0 + r^{\mathbf{P_A}} + x_0y_1 + r^{\mathbf{P_B}}$
$\mathbf{P_B}$	x_0	y_1	x_0y_1	$x_0y_0 + r^{\mathbf{P_A}} + x_0y_1 + r^{\mathbf{P_B}}$
$\mathbf{P_C}$	x_1	y_0	x_1y_0	$x_1y_0 + r^{\mathbf{P_C}} + x_1y_1 + r^{\mathbf{P_D}}$
$\mathbf{P_D}$	x_1	y_1	x_1y_1	$x_1y_0 + r^{\mathbf{P_C}} + x_1y_1 + r^{\mathbf{P_D}}$

TABLE 4: Multiplication given two shared values with inconsistent share modes

value x_0, $\mathbf{P_C}$ and $\mathbf{P_D}$ hold the same value x_1 such that $x = x_0 + x_1$. We define $[\cdot]^{\mathbf{P_i}}_{\mathsf{AC}}$ to be the share value of $\mathbf{P_i}$.

- $[\cdot]_{\mathsf{AB}}$-sharing : We say that a value x is $[\cdot]_{\mathsf{AB}}$-shared among parties $\{\mathbf{P_i}\}$, if $\mathbf{P_A}$ and $\mathbf{P_C}$ hold the same value x_0, $\mathbf{P_B}$ and $\mathbf{P_D}$ hold the same value x_1 such that $x = x_0 + x_1$. Same as above, we denote the share of $\mathbf{P_i}$ as $[\cdot]^{\mathbf{P_i}}_{\mathsf{AB}}$.

We denote the share-mode as ψ, ϕ, θ, with $\psi, \phi, \theta \in \{\mathsf{AC}, \mathsf{AB}\}$. We say that a value x is $[\cdot]_{\mathsf{4o4}}$-shared among parties $\{\mathbf{P_i}\}$, if $\mathbf{P_i}$ hold share x_i respectively such that $x = \sum x_i$.

At the first glance, this seems to be the same as 2-out-of-2 sharing in 2PC that is simply re-used to 4PC setting. But the most attractive part of $\mathcal{X}$-share is that now we are able to eliminate most of the heavy local computations in 2PC and replicated-sharing-based 3PC, with reduced or equal communication volume of each party (See Fig. 2).

4.1.2. Linearity. If the share-modes of both shared values are identical, it is easy to observe that the linear computations can be executed locally with $\mathcal{X}$-share. Given $[\cdot]_{\mathsf{AC}}$-sharing (or $[\cdot]_{\mathsf{AB}}$-sharing) of secret values x, y and public constants e_0, e_1, parties can locally compute $e_0[x]_{\mathsf{AC}} + e_1[x]_{\mathsf{AC}}$. The trick continues when parties have to compute $[x]_{\mathsf{AC}} + e_2$, where e_2 is a public constant.

Now we consider the case if the share modes of secret x and secret y are different, e.g. $[x]_{\mathsf{AC}}$ and $[y]_{\mathsf{AB}}$. In order to keep the output to maintain either $[\cdot]_{\mathsf{AC}}$-sharing or $[\cdot]_{\mathsf{AB}}$-sharing, parties have to jointly change the share mode of y (or x) by executing Π_{chMode} (see §4.1.4), then locally compute $[x]_{\mathsf{AC}} + [y]_{\mathsf{AC}}$.

4.1.3. 4PC Multiplication. The most important application of $\mathcal{X}$-share is 4PC multiplication. We begin with computing $[z]_{\mathsf{4o4}} = [x]_\psi[y]_\phi$, where $\psi \neq \phi$. To perform the multiplication of two secret values, parties have to jointly compute:

$$\begin{aligned} xy &= (x_0 + x_1)(y_0 + y_1) \\ &= x_0y_0 + x_0y_1 + x_1y_0 + x_1y_1 \end{aligned}$$

Suppose the secret value x is $[\cdot]_{\mathsf{AC}}$-shared and the secret value y is $[\cdot]_{\mathsf{AB}}$-shared (or reversely), each party can locally compute exactly one out of four terms shown in the above equation. This yields a 4-out-of-4 sharing $[z]_{\mathsf{4o4}} = [x]_{\mathsf{AC}}[y]_{\mathsf{AB}}$. For further computations, parties send their own masked share $[z]^{\mathbf{P_i}}_{\mathsf{4o4}} + r^{\mathbf{P_i}}$ to their reshare partner, where $\sum r^{\mathbf{P_i}} = 0$. Since each *zero sharing* is fresh, parties can freely choose to rebuild either $[z]_{\mathsf{AC}}$ or $[z]_{\mathsf{AB}}$ according

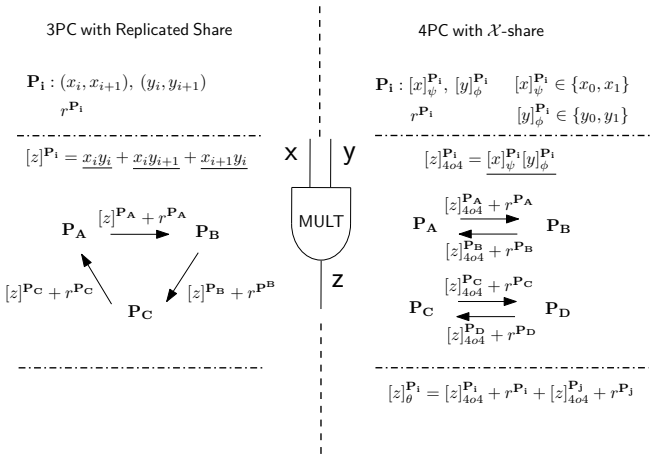


Fig. 2: Demonstration of $\mathcal{X}$-share compared with other share constructions in multiplication protocol.

to the incoming computations. An illustration of a local share distribution and computation is provided in Table. 4.

Due to the fact that we are using fixed-point numbers to represent both x and y, the re-shared result z has to be truncated to maintain the p decimal bit precision. Remark that after re-sharing, both $[z]_{\mathsf{AC}}$ and $[z]_{\mathsf{AB}}$ yields a 2-out-of-2 sharing, thus we are free to apply the truncation technique Π_{trunc} introduced by secureML [55] to avoid the additional communication overhead and round within the truncation protocols Π_{trunc1} and Π_{trunc2} proposed by ABY3 [54]. A detailed description of our multiplication protocol is shown in Fig. 3. We further provide an overview in Fig. 2 of our 4PC multiplication compared with 3PC, regarding size of local shares, communication and local computation.

In contrast to linear operation, an unwilling situation for multiplication is when the share-modes of both secrets x and y are identical. Parties have to execute the Π_{chMode} (Fig. 5) to change the share-mode of either x or y (not both) before multiplication.

4.1.4. Change Share-mode. Here we present the protocol Π_{chMode} for changing share-modes. We first define correlated randomness called *changeM sharing* or shortly CMS, denoted as $[\![r]\!]_{\psi\mathsf{to}\phi}$:

- $[\![r]\!]_{\mathsf{ACtoAB}}$: We say that a randomness r is $[\![\cdot]\!]_{\mathsf{ACtoAB}}$-shared among parties $\{\mathbf{P_i}\}$, if $\mathbf{P_A}$ and $\mathbf{P_C}$ hold separately r_0 and r_1, while $\mathbf{P_B}$ and $\mathbf{P_D}$ both hold r, such that $r = r_0 + r_1$. We define $[\![r]\!]^{\mathbf{P_i}}_{\mathsf{ACtoAB}}$ to be the share value of $\mathbf{P_i}$.
- $[\![r]\!]_{\mathsf{ABtoAC}}$: We say that a randomness r is $[\![\cdot]\!]_{\mathsf{ABtoAC}}$-shared among parties $\{\mathbf{P_i}\}$, if $\mathbf{P_A}$ and $\mathbf{P_B}$ hold separately r_0 and r_1, while $\mathbf{P_C}$ and $\mathbf{P_D}$ both hold r, such that $r = r_0 + r_1$. We define $[\![r]\!]^{\mathbf{P_i}}_{\mathsf{ABtoAC}}$ to be the share value of $\mathbf{P_i}$.

Suppose parties are willing to change share-mode of x from $[\cdot]_{\mathsf{AC}}$-sharing to $[\cdot]_{\mathsf{AB}}$-sharing, we require parties to already hold $[\![r]\!]_{\mathsf{ACtoAB}}$ after the pre-processing phase. During the execution of Π_{chMode}, $\mathbf{P_A}$ and $\mathbf{P_C}$ simply exchange their

Protocol Π_{Mult}

Private inputs: Parties hold $[x]_\psi$, $[y]_\phi$, where $\psi \neq \phi$.
Public inputs: Public parameters, θ.
Outputs: $[z]_\theta$ with $z = xy$.
Preprocessing:

- $\mathbf{P_i}$ sends $(\mathsf{ZeroSGen}, \mathbf{P_i}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$, receives $r^{\mathbf{P_i}}$ as output.

Protocols:

1. Parties locally compute $[z]_{\mathsf{4o4}} = [xy]_{\mathsf{4o4}}$, where $[z]_{\mathsf{4o4}}^{\mathbf{P_i}} := [x]_\psi^{\mathbf{P_i}}[y]_\phi^{\mathbf{P_i}}$.
2. We denote $\mathbf{P_j}^{\mathbf{P_i}}$ as $\mathbf{P_i}$'s reshare partner.
   - If $\theta = \mathsf{AC}$: $\mathbf{P_A}$ and $\mathbf{P_C}$ set $\mathbf{P_j}^{\mathbf{P_A}} = \mathbf{P_C}$ and $\mathbf{P_j}^{\mathbf{P_C}} = \mathbf{P_A}$, respectively. $\mathbf{P_B}$ and $\mathbf{P_D}$ set $\mathbf{P_j}^{\mathbf{P_B}} = \mathbf{P_D}$ and $\mathbf{P_j}^{\mathbf{P_D}} = \mathbf{P_B}$, respectively.
   - If $\theta = \mathsf{AB}$: $\mathbf{P_A}$ and $\mathbf{P_B}$ set $\mathbf{P_j}^{\mathbf{P_A}} = \mathbf{P_B}$ and $\mathbf{P_j}^{\mathbf{P_B}} = \mathbf{P_A}$, respectively. $\mathbf{P_C}$ and $\mathbf{P_D}$ set $\mathbf{P_j}^{\mathbf{P_C}} = \mathbf{P_D}$ and $\mathbf{P_j}^{\mathbf{P_D}} = \mathbf{P_C}$, respectively.
3. Each $\mathbf{P_i}$ computes $e^{\mathbf{P_i}} := [z]_{\mathsf{4o4}}^{\mathbf{P_i}} + r^{\mathbf{P_i}}$ and sends $e^{\mathbf{P_i}}$ to its reshare partner $\mathbf{P_j}$.
4. Upon receiving $e^{\mathbf{P_j}}$ from $\mathbf{P_j}^{\mathbf{P_i}}$, $\mathbf{P_i}$ sets $[z]_\theta^{\mathbf{P_i}} := e^{\mathbf{P_i}} + e^{\mathbf{P_j}}$.

Fig. 3: Four party multiplication protocol

own 2-out-of-2 sharing masked with r_0 and r_1, obtaining their new share $x_0 + x_1 - r_0 - r_1$, while $\mathbf{P_B}$ and $\mathbf{P_D}$ set their shares to be r locally. This yields a fresh $[x]_{\mathsf{AB}}$. The CMS can be generated by computing $\mathsf{PRF}()$ with pre-shared keys in the pre-processing stage. We formally define our CMS generation protocol Π_{CMSGen} in Fig. 4, as well as the online protocol Π_{chMode} in Fig. 5.

4.1.5. Division. If parties have to jointly divide a public value γ which is not a power of two, they cannot simply divide this public value locally, since this operation might remove the carry bit of the shared values and thus leads to a incorrect result. We use the truncation protocol Π_{trunc2} in [54] as a division protocol Π_{Div}, which consumes a correlated randomness that we call division share $([r]_\psi, [r']_\psi)$, where $[r']_\psi = [r/\gamma]_\psi$. The idea behind this protocol is to first reveal the shared value $[x]_\psi$ masked with $[r]_\psi$. Parties can compute publicly $(x - r)/\gamma$ then unmask this value by computing $(x - r)/\gamma + [r']_\psi$ locally.

4.2. Boolean Computation

This is the special case for $\ell = 1$ in $\mathbb{Z}_{2^\ell}$. The linearity preserves and parties can simply replace all additions (and subtractions) with XORs and multiplication with ANDs when executing boolean operations.

4.3. Share Conversion

For PPML, non-linear functions (such as ReLU, max-pooling etc.) can be evaluated more appropriate with MPC

Protocol Π_{CMSGen}

Private inputs:
$\mathbf{P_A}$ holds k^0_{ACtoAB}, k^1_{ACtoAB}, k^0_{ABtoAC} and k^1_{ABtoAC}.
$\mathbf{P_B}$ holds k^0_{ACtoAB}, k^2_{ACtoAB}, k^1_{ABtoAC} and k^2_{ABtoAC}.
$\mathbf{P_C}$ holds k^1_{ACtoAB}, k^2_{ACtoAB}, k^0_{ABtoAC} and k^2_{ABtoAC}.
$\mathbf{P_D}$ holds k^0_{ACtoAB}, k^2_{ACtoAB}, k^0_{ABtoAC} and k^2_{ABtoAC}.

Public inputs: Public parameters, sid, $\psi\mathsf{to}\phi$.
Outputs: $[\![r]\!]_{\psi\mathsf{to}\phi}$.
Protocols:

- If $\psi\mathsf{to}\phi = \mathsf{ACtoAB}$:
  1. $\mathbf{P_A}$ computes $r_0 := \mathsf{PRF}_{k^0_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k^1_{\mathsf{ACtoAB}}}(\mathsf{sid})$ then sets $[\![r]\!]^{\mathbf{P_A}}_{\mathsf{ACtoAB}} := r_0$.
  2. $\mathbf{P_C}$ computes $r_1 := \mathsf{PRF}_{k^1_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k^2_{\mathsf{ACtoAB}}}(\mathsf{sid})$ then sets $[\![r]\!]^{\mathbf{P_C}}_{\mathsf{ACtoAB}} = r_1$.
  3. $\mathbf{P_B}$ and $\mathbf{P_D}$ compute: $r := \mathsf{PRF}_{k^0_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k^2_{\mathsf{ACtoAB}}}(\mathsf{sid})$ and set $[\![r]\!]^{\mathbf{P_B}}_{\mathsf{ACtoAB}} = [\![r]\!]^{\mathbf{P_D}}_{\mathsf{ACtoAB}} := r$, respectively.
- Otherwise if $\psi\mathsf{to}\phi = \mathsf{ABtoAC}$:
  1. $\mathbf{P_A}$ computes $r_0 := \mathsf{PRF}_{k^0_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k^1_{\mathsf{ACtoAB}}}(\mathsf{sid})$ then sets $[\![r]\!]^{\mathbf{P_A}}_{\mathsf{ACtoAB}} := r_0$.
  2. $\mathbf{P_B}$ computes $r_1 := \mathsf{PRF}_{k^1_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k^2_{\mathsf{ACtoAB}}}(\mathsf{sid})$ then sets $[\![r]\!]^{\mathbf{P_B}}_{\mathsf{ACtoAB}} = r_1$.
  3. $\mathbf{P_C}$ and $\mathbf{P_D}$ compute: $r := \mathsf{PRF}_{k^0_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k^2_{\mathsf{ACtoAB}}}(\mathsf{sid})$ and set $[\![r]\!]^{\mathbf{P_C}}_{\mathsf{ACtoAB}} = [\![r]\!]^{\mathbf{P_D}}_{\mathsf{ACtoAB}} := r$, respectively.

Fig. 4: Four party *changeM share* generation protocol

Protocol Π_{chMode}

Private inputs: Parties hold $[x]_\psi$.
Public inputs: Public parameters, ϕ.
Outputs: $[x]_\phi$.
Preprocessing:

- $\mathbf{P_i}$ sends $(\mathsf{CMSGen}, \psi\mathsf{to}\phi\ \mathbf{P_i}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$, receives $[\![r]\!]^{\mathbf{P_i}}_{\psi\mathsf{to}\phi}$ as output.

Protocols:

- If $\psi = \mathsf{AC}$ and $\phi = \mathsf{AB}$:
  1. $\mathbf{P_A}$ computes $d_0 := [x]^{\mathbf{P_A}}_\psi - [\![r]\!]^{\mathbf{P_A}}_{\psi\mathsf{to}\phi}$, then sends d_0 to $\mathbf{P_C}$.
  2. $\mathbf{P_C}$ computes $d_1 := [x]^{\mathbf{P_C}}_\psi - [\![r]\!]^{\mathbf{P_C}}_{\psi\mathsf{to}\phi}$, then sends d_1 to $\mathbf{P_A}$.
  3. Upon receiving d_0 and d_1, $\mathbf{P_A}$ and $\mathbf{P_C}$ set $[x]^{\mathbf{P_A}}_\phi := d_0 + d_1$, $[x]^{\mathbf{P_C}}_\phi := d_0 + d_1$ respectively.
  4. $\mathbf{P_B}$ and $\mathbf{P_D}$ set $[x]^{\mathbf{P_B}}_\phi := [\![r]\!]^{\mathbf{P_B}}_{\psi\mathsf{to}\phi}$, $[x]^{\mathbf{P_D}}_\phi := [\![r]\!]^{\mathbf{P_D}}_{\psi\mathsf{to}\phi}$ respectively.
- Otherwise if $\psi = \mathsf{AB}$ and $\phi = \mathsf{AC}$, switch the role of $\mathbf{P_C}$ with $\mathbf{P_B}$, do the same as above.

Fig. 5: Four party change share mode protocol

protocols over boolean inputs [42, 54, 59, 67, 73], while other linear functions (multiplication, convolutions etc.) prefer arithmetic shared values. In following we show how conversion between sharing types works, and how parties

can determine the share-mode of outputs.

4.3.1. $\mathcal{X}$-dabit. As an important building block, we extend edabit introduced by Escudero *et al.* [22] to $\mathcal{X}$-dabit. Here $b \xleftarrow{\$} \mathbb{Z}_{2^\ell}$, and ψ and ϕ can be identical.

$$\mathcal{X}\text{-dabit} := ([b]_\psi, \langle b \rangle_\phi := (\langle b_0 \rangle_\phi, \cdots, \langle b_{l-1} \rangle_\phi))$$

To generate $\mathcal{X}$-dabit, the four parties are assigned into two groups of size two. Then following the protocols proposed by [22] for 2PC setting inside each group, parties ends up holding the same randomness and generate shares in both arithmetic and boolean worlds. This allows parties to generate $([b]_\psi, \langle b \rangle_\phi)$, where $\psi = \phi$. To change the share-mode of either $[b]_\psi$ or $\langle b \rangle_\phi$, parties run Π_{chMode} (Fig. 5).

4.3.2. Arithmetic vs. Boolean. We first consider one bit case ($\ell = 1$), where parties have to convert $[x]_\psi$ to $\langle x \rangle_\phi$ with $x \in \mathbb{Z}_2$ (a B2A protocol for one single bit). Note that in this case, parties sample $b \xleftarrow{\$} \mathbb{Z}_2$ in $\mathcal{X}$-dabit. The boolean share of this $\mathcal{X}$-dabit becomes one-bit share among parties. By using such an $\mathcal{X}$-dabit, parties simply open their local shares $[x]_\psi^{\mathbf{P_i}}$ masked with $[b]_\psi^{\mathbf{P_i}}$, then locally unmask the revealed value $x - b$ with $\langle b \rangle_\phi$. Note that the unmasking computation could be applied locally due to its linearity. Converting $\langle x \rangle_\phi$ to $[x]_\psi$ works in the same manner vice versa. A detailed protocol description is in Fig. 6.

If $x \in \mathbb{Z}_{2^\ell}$ with $\ell > 1$, parties then generate an $\mathcal{X}$-dabit with more than one bit in boolean share to support the conversion protocol.

Protocol 4: Bit to Arithmetic Π_{BitToA}

Private inputs: Parties hold $\langle x \rangle_\phi$, where $x \in \{0, 1\}$.
Public inputs: Public parameters, ψ.
Outputs: $[x]_\psi$.
Preprocessing:

- $\mathbf{P_i}$ sends $(\mathcal{X}\text{-dabitGen}, \psi, \phi, \ell^{\mathsf{B}}, \mathbf{P_i}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$ with $\ell^{\mathsf{B}} = 1$, receives $([b]_\psi^{\mathbf{P_i}}, \langle b \rangle_\phi^{\mathbf{P_i}})$ as output.

Protocols:

1. Parties locally compute and then reveal $h := \langle x \rangle_\phi \oplus \langle b \rangle_\phi$, where $h \in \{0, 1\}$.
2. If $h = 0$, parties set $[x]_\psi = [b]_\psi$, otherwise $[x]_\psi = 1 - [b]_\psi$.

Fig. 6: Four party bit to arithmetic protocol

4.4. Comparison

We now introduce our secure 4PC comparison protocol Π_{Comp}. Using the same technique mentioned in [73], parties firstly reveal $[x]_\psi$ by masking it with $[b]_\psi$ (arithmetic part of an $\mathcal{X}$-dabit). Now parties hold $\langle b \rangle_\psi$ (boolean part of an $\mathcal{X}$-dabit) and a public revealed $x - b$ over $\mathbb{Z}_{2^\ell}$. After computing the bit decomposition of $x - b$, parties will jointly compute a PPA circuit to securely extract the sign bit of $[x]_\psi$. To do so, parties will prepare a shared propagator $\langle p \rangle_\psi = \langle x - b \rangle_\psi \oplus \langle b \rangle_\psi$ and a shared generator $\langle g \rangle_\phi = \langle x - b \rangle_\phi * \langle b \rangle_\phi$, where $A * B$ denotes a bit-wise AND of A and B, and $\psi \neq \phi$. To prepare such a $\langle g \rangle_\phi$, parties call Π_{chMode} once before computing the PPA. In return now 50% of the secure AND protocols are already executable in an efficient 4PC way. For the rest of AND computations, we choose to let parties call Π_{chMode} once in each round to change the share-mode of the updated propagator. We refer readers to §5.3 for more details.

5. Efficient PPML Implementation

For further discussion, we highlight all technical details and advantages given $\mathcal{X}$-share for each layer implementation in CNN. We compare the necessary computation and communication effort of our framework with existing ones in 2PC, 3PC and 4PC. We show that our framework achieves already optimized local computation overhead for layers such as Convolution and Fully connected Layer §5.1. Furthermore, parties benefit from holding $\mathcal{X}$-share, which enables them to locally rescale the output of multiplications §5.2. In the end of this section, we provide a detailed description about our activation layer implementation §5.3.

5.1. Convolution and FC

We use $matmul$ for both element multiplication and matrix multiplication (convolution) for conciseness. A summary of Force and existing works in multiplication at each party is showed in Table 2.

CrypTen [42] implements 4PC protocols with 4-out-of-4 sharing. The $matmul$ needs Beaver triples, which yields three local $matmul$ operations, two elements/matrices sending (to each peer) and six elements/matrices receiving at each party. Let the bit-length of the element/matrix be ℓ. While peer-to-peer channel is used, a total of $2 \times 3 + 6 = 12\ell$ data volume has to be exchanged. We point out that even using a broadcast channel mentioned in paper, each party still has 8ℓ to send/receive.

P-FantasticFour [17, 73] on the other hand, uses replicated share over four shares, which greatly reduces the communication volume compared to CrypTen [42]. However, even such an optimization still results 4ℓ communication overhead for each party per $matmul$.

As already mentioned in §4.1.3 and Fig. 2, 3PC needs the re-sharing protocol which requires one round communication, where each party needs 2ℓ. Thus, even compared to the improved protocols designed by ABY2.0 [59], protocols proposed for 3PC with replicated sharing (such as ABY3 [54], Piranha [73]) achieves already the same communication effort and much simpler local computation.

Although 3PC multiplication seems to dominate 2PC in communication, we still observe a huge computational complexity reduction and a much simplified connection channel establishment given $\mathcal{X}$-share in Force for 4PC. First of all, parties only have to compute one single $matmul$ locally, eliminating 66% of local computation (see the underlined parts in Fig. 2). And in fact, parties exchange their local

TABLE 5: Truncation cost. Comparison of Force and existing works against **semi-honest** adversary regarding Π_{trunc} only and combined with Π_{Mult}. Communication volume is calculated for end-to-end channels.

Setting	Ref.	Trunc only		Comb. with Mult	
		Comm	Rounds	Comm	Rounds
2PC	P-SecureML[55]	0	0	4ℓ	1
3PC	CryptGPU[67]	ℓ	1	3ℓ	2
	P-Falcon [73]	2ℓ	1	4ℓ	1
4PC	CrypTen[42]	4ℓ	1	12ℓ	2
	P-FantasticFour[73][17]	2ℓ	1	6ℓ	2
	Our **Force**	**0**	**0**	$\mathbf{2\ell}$	**1**

shares with one single partner instead of two in 3PC setting, which yields a simpler peer-to-peer connection.

5.2. Truncation

As already mentioned in §4.1, parties have to rescale the shared output of $matmul$ for consistence in precision. In 2PC, SecureML [55] has proposed a local truncation technique, such that parties simply truncate the last p bits without any interaction. However, ABY3 [54] has proven that such technique only supports the 2-out-of-2 sharing and fails in replicated sharing in 3PC setting. Instead, all three parties perform Π_{trunc2} with the help of a precomputed truncation share $([r], [r'])$, where $[r'] = [r/2^p]$ (also known as a division share with the public value $\gamma = 2^p$). This protocol could be separately executed after resharing protocol, requiring one additional communication round. Or it can be performed combined with the resharing protocol, where parties reshare and truncate the 3-out-of-3 result in a single round exchanging 4ℓ information overall. We call such a protocol Π_{RsTrunc2}. Besides, CryptGPU [67] chooses to implement another truncation protocol Π_{Trunc1} proposed by ABY3 [54] to skip the preprocessing stage generating truncation share. As already pointed out, Π_{Trunc1} requires two rounds and 3ℓ bits communication volume totally.

CrypTen [42] on the other hand, requires parties to execute multiplication and truncation in two separate rounds, where its truncation protocol results at least a total 6ℓ bits communication volume (4ℓ bits volume based on a broadcast channel). The idea was inspired by SecureNN [70].

P-FantasticFour [17, 73] extends the same technique proposed by ABY3 to a four party version, which yields a 2ℓ communication overhead for each party and an additional round.

Regardless, neither Π_{RsTrunc2} is communication-free in the truncation part. One of our main contribution is, that our $\mathcal{X}$-share is compatible with the local truncation technique. We summarize the truncation cost of all systems in Table 5.

5.3. Activation Layer

5.3.1. Our Carryout Implementation. One of Piranha's main contribution is the iterator-based in-place carryout implementation, which cuts the peak memory load in half. We illustrate their procedure in Fig. 7. Notably, even though

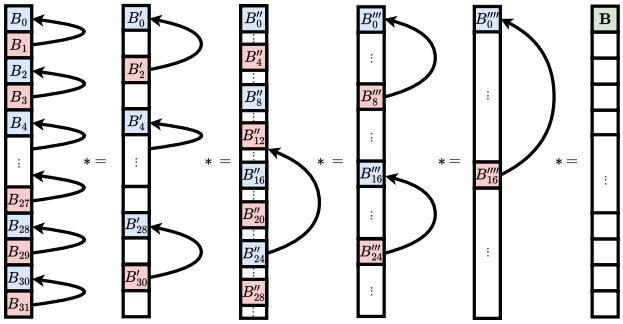


Fig. 7: An illustration of Piranha's in-place carryout implementation for a 32-bit value. `B` means `Byte`.

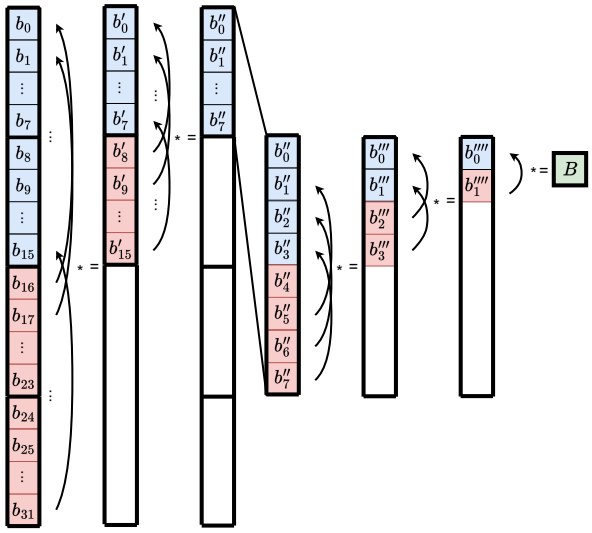


Fig. 8: An illustration of our Force's in-place carryout implementation for a 32-bit value. `b` means `bit`. A bold border points to one byte memory.

they compute for a 32-bit value, they decompose it into 32 Bytes. Then perform the operation between even bytes and odd bytes in a tree order. The total communication volume is $16 + 8 + 4 + 2 + 1 = 31$ Bytes.

To be more memory-efficient, we decompose the 32-bit value into 4 Bytes, reducing the memory consumption by c.a. 87.5%. Different from Piranha, we perform the operation between higher bytes and lower bytes. When we reach 1 Byte or 8 bits, we use bit operation within that byte. The process is showed in Fig. 8. The total communication volume of our Force is $2 + 1 + 1 + 1 + 1 = 6$ Bytes.

5.3.2. Secure Comparison. The standard ReLU activation function requires parties to execute a compare protocol to extract the sign bit of an originally arithmetically shared value.

ABY2.0 [59] chooses to compute PPA introduced in [29] for boolean addition, which results total $\log \ell + 1$ rounds. As a framework, CrypTen [42] provides two options to eval-

uate a PPA circuit. For memory-efficient approach, parties will have to sequentially add their boolean shares together ($n \log \ell$ rounds), while executing a memory-inefficient approach requires $\log n \log \ell$ rounds.

Adaptively, our Π_{Comp} can be designed either in a mixed 2PC and 4PC way, or completely in a 4PC way. An implementation with both 2PC and 4PC protocol executions achieves total $\log \ell + 2$ rounds, which requires slightly one more round compared to protocols in 2PC setting and 3PC setting, but dominates $2 \log \ell$ rounds needed in CrypTen in 4PC setting. Even so, we choose to let parties only execute 4PC protocols during Π_{Comp}. This results total $2\log \ell + 2$ rounds, since execution of Π_{chMode} in each computation round of a PPA doubles the communication rounds to compute the carry out. As compensation, parties save at least 25% of local computations compared to a mixed protocol executions, and meanwhile the communication volume stays the same. To our experiments, Π_{Comp} and its implementation already achieves a huge speed up compared to other settings, see Section 6.4 for more details.

5.4. Accelerated Backward in Training

In this section we introduce our accelerated implementation of backward phase. So far, we have discussed the improvement of our framework comparing to others in each single layer, and in particular during forward phase. A more complex scenario occurs in the backward phase: for example, parties hold a shared x in $[\cdot]_{\mathsf{AC}}$-sharing, which has to be multiplied with two shared values y in $[\cdot]_{\mathsf{AC}}$-sharing and y' in $[\cdot]_{\mathsf{AB}}$-sharing.

First of all, we exclude this situation from forward phase, since no shared value has to be reused in different computation. The easiest way to implement the backward phase is to let parties execute Π_{chMode} if needed. Such a naive solution results an extra round and communication overhead, but it already has a huge performance improvement comparing to other frameworks in our early stage. Fortunately, we found out that the most efficient way is to let parties hold one shared value in both share-modes, which then enables parties to perform 4PC computations everywhere during backward phase. Such critical values are normally only weights in each layer, meaning that parties are capable to trade a small portion of their GPU memory benefit for a huge computation acceleration. Remark that holding a shared value in both share-modes does not leak any information to parties, since local shares of each shared value in different share-modes will be chosen freshly (e.g. $x = x_0 + x_1$ and $x = x'_0 + x'_1$).

6. Evaluations

We build Force on top of Piranha [73], which itself is mainly inspired by Falcon [71]. All these are implemented in C++. To differentiate from them, we introduce a new 4-party sharing type $\mathcal{X}$-share, and make it support all relevant PPML operations. In this section, we will thoroughly evaluate $\mathcal{X}$-share and answer the following questions:

1) *In Comparison to state-of-the-art PPML prior work, how efficient $\mathcal{X}$-share is when performing secure inference and secure training of neural networks (§6.2 and §6.3)?*
2) *How well does $\mathcal{X}$-share accelerate common computation tasks, for linear operations such as convolution and matrix multiplication (§6.4.1) and non-linear operations like ReLU (§6.4.2)?*
3) *How many resources does $\mathcal{X}$-share consume? More specifically, what are the requirements for network (§6.5) and graphic memory (§6.6)?*
4) *Does PPML with $\mathcal{X}$-share perfectly reproduce the expected results as in plaintext ML? Is the computation accurate (§6.7)?*

6.1. Evaluation Setup

6.1.1. Testbed Environment. We run our evaluations on cloud servers. The servers come with 2 CPUs, `Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz`, and 12×128GB of RAM. CPU and RAM are only relevant for the evaluation of Cheetah [34]. All the other evaluations mainly concern GPU and VRAM. Each of our servers is equipped with one GPU, `NVIDIA Tesla P100-PCIE`, which includes 16GB of dedicated VRAM. We consider two types of network environments. One is the LAN setting with 10Gbps bandwidth and $0.2ms$ round-trip latency. The other is the WAN setting with 100Mbps bandwidth and $40ms$ round-trip latency. We simulate these two network settings by using the `tc` tool [2].

Our server is running `Ubuntu 18.04.6 LTS`. As for GPU support, we use `CUDA 10.1.243`. We do not use the latest `CUDA` due to compatibility issues with `PyTorch`, considering that both CryptGPU [67] and CrypTen [42] rely on `PyTorch`. Our implementation is based on Piranha [73] [3], at commit $bd9c8c4$. Except for our new sharing type $\mathcal{X}$-share, we integrate support for batch normalization, while the original Piranha [73] only supports layer normalization. We also add extra support for complex `ResNet`, including `ResNet50, ResNet101, ResNet152`. In Piranha, there is only basic block for `ResNet18`.

6.1.2. Baseline. We choose as baselines several state-of-the-art systems that have **semi-honest security**. They are summarized in Table 1.

For 2PC, Cheetah [34] is the most recent PPML work using FHE and correlated oblivious transfer (cOT), which is completely different from ours. They declare to be the most efficient on CPU. We run it on the same server as a baseline of CPU-based PPML. SecureML [55] is also CPU-based and the only 2-party system supporting both private inference and private training. Piranha [73] ports it to GPU and states to outperform the original version. Thus we only use the new version as a baseline and refer to it as P-SecureML.

2. https://man7.org/linux/man-pages/man8/tc.8.html
3. https://github.com/ucbrise/piranha/

TABLE 6: Summary of tested neural network models concerning different datasets. Layers for small and medium datasets may slightly vary as mentioned in Section 6.1.3.

Model	Number of Layers						Number of Parameters		
	Conv	ReLU	FC	Pool	BN	Total	CIFAR10	Tiny ImageNet	
AlexNet	5	7	3	3	0	18	3.9M	6.1M	35.9M
VGG16	13	15	3	6	0	37	14.9M	15.3M	54.5M
ResNet152	155	151	1	2	155	464	58.1M	58.6M	60.2M

For both 3-party and 4-party, we only consider the **honest-majority** setting. Concerning the 3-party setting, Falcon [71] is the fastest on CPU. Piranha [73] largely improves the performance by porting it to GPU. We mark it as P-Falcon. Even though P-Falcon is fast most of the time, it loses to CryptGPU [67] on some large datasets. We include both of them as baselines. CryptGPU [67] is deployed with the latest Github source code [4], at commit $2ff57b2$.

As for 4-party, CrypTen [42] is the only one which has **semi-honest security** in an honest-majority setting. We deploy it using their latest Github source code [5], at commit $efe8eda$. All the other 4-party or more-party systems are for **malicious** adversaries, which are slowed down by heavy verification or validations [3, 18, 26, 33, 36, 65]. For fairness, we should not compare with them. Yet, for better insight, we include the benchmarks of GPU version of FantasticFour [17], re-implemented by Piranha [73] for **semi-honest security**, which we call P-FantasticFour. Considering that we completely outperform P-FantasticFour in all settings, Force would only be much faster than the actual maliciously secure FantasticFour.

We run all the evaluations with 20 bits of fixed-point precision. The calculations are over the 64-bit ring $\mathbb{Z}_{2^{64}}$, except Cheetah [34], which supports maximum 44-bit. All the experiments are performed multiple times. Then we calculate the benchmarks by averaging all the results except the first run, to mitigate the influence of system initialization and runtime randomness.

6.1.3. Models and Datasets. We consider three datasets in different sizes for our evaluations:

- Small dataset: CIFAR10[45], $60,000$ 32×32 RGB images in 10 classes.
- Medium dataset: TinyImageNet[48] (Tiny for short), $100,000$ 64×64 RGB images in 200 classes.
- Large dataset: ImageNet[64], $1,000,000$ 224×224 RGB images in $1,000$ classes.

These datasets are evaluated in three neural network models of different depths:

- Shallow model: AlexNet[46], an 8-layer convolutional network.
- Medium model: VGG16[66], a 16-layer convolutional network.
- Deep model: ResNet152[31], a 152-layer convolutional network.

4. https://github.com/jeffreysijuntan/CryptGPU
5. https://github.com/facebookresearch/CrypTen

We try to keep the models as much as they are in their original publications. However, due to the various input sizes of different datasets, as well as performance considerations, we slightly adjust the structure similarly to CryptGPU [67] and Falcon [71]. We summarize the typical number of layers and number of parameters in Table 6.

6.2. Secure Inference of Neural Networks

In Table 7, we list Force's total running time of an inference pass for all datasets and models described in §6.1.3 in *LAN* setting. Our Force completely outperform all the baseline systems in all evaluations.

Firstly, we notice that in current *LAN* setting, the state-of-the-art CPU-based Cheetah is slower than all other GPU-based systems in all experiments. To our evaluation, the performance of Cheetah is comparable with those of GPU-based frameworks in the case of small datasets and shallow models. For example, they can be as good as 2 times slower in CIFAR10+AlexNet. However, concerning big datasets and medium or deep models like ImageNet+VGG16, they are at least 100 times slower.

When comparing all GPU-based systems, the C++-implemented (P-SecureML, P-Falcon, P-FantasticFour and Force) perform much better than the Python-implemented (CryptGPU and CrypTen). This could result from the language performance difference. CryptGPU is faster than CrypTen most of the time, except on ResNet152. The main reason may be that newer PyTorch involves an optimization called *Fused Batch Normalization*, which only affects ResNet152 in our evaluations.

With the acceleration brought by our novel sharing type $\mathcal{X}$-share, Force beats the other three Piranha-based systems, P-SecureML, P-Falcon and P-FantasticFour, in all experiments. Force is at least 3.1 times faster than P-SecureML, at least 2 times faster than P-Falcon and at least 5 times faster than P-FantasticFour.

The results are a bit different in *WAN* setting, as showed in Table 8. Yet, our Force still outperforms all the other systems. Cheetah sometimes does better than CryptGPU and CrypTen, benefiting from fewer communication rounds. On large datasets and deep models such as ImageNet + VGG16, CryptGPU is faster than P-Falcon, due to lower communication volume.

6.3. Secure Training of Neural Networks

Under the same setting as in §6.2, we list Force's total running time of a training pass for all datasets and models in Table 10. Considering that Cheetah does not support private training, it is omitted here. Again, our Force completely outperform all the baseline systems in all evaluations.

Python-implemented (CryptGPU and CrypTen) still perform worse than the C++-implemented systems. The performance gap between CryptGPU and CrypTen gets larger compared to the inference pass. CryptGPU could even be 7.4 times faster than CrypTen, when training CIFAR10 on

TABLE 7: Running time (Second) of an ***inference*** pass in ***LAN*** setting with BatchSize $= 1$.

	CIFAR10			Tiny			ImageNet		
	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152
P-SecureML	0.41	1.48	7.89	0.55	2.19	9.44	2.50	15.70	31.46
CryptGPU	1.15	2.91	35.58	1.14	3.83	38.21	2.42	12.74	49.54
P-Falcon	0.29	0.89	5.18	0.35	1.37	6.24	1.12	10.03	20.39
CrypTen	1.05	3.48	26.04	1.25	5.20	29.10	4.59	32.75	62.58
P-FantasticFour	0.72	2.20	12.81	0.87	3.40	15.59	2.72	24.03	49.74
Force	**0.12**	**0.35**	**2.54**	**0.14**	**0.54**	**3.01**	**0.43**	**3.26**	**9.70**
Cheetah	2.67	80.43	66.96	19.74	325.30	263.87	383.97	4026.87	3226.62
PyTorch	0.0008	0.0017	0.0264	0.0009	0.0017	0.0266	0.0009	0.0017	0.0268

TABLE 8: Running time (Second) of an ***inference*** pass in ***WAN*** setting with BatchSize $= 1$.

	CIFAR10			Tiny			ImageNet		
	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152
P-SecureML	12.20	57.54	239.81	21.64	121.05	241.73	179.19	1126.83	1907.65
CryptGPU	18.41	44.17	807.32	19.46	65.15	846.11	48.53	359.46	1387.28
P-Falcon	2.85	11.08	91.06	3.80	28.27	119.33	30.79	370.70	730.97
CrypTen	34.37	103.26	721.67	43.47	256.77	876.92	397.49	2203.29	4649.98
P-FantasticFour	7.60	41.39	218.33	13.00	125.80	368.82	135.93	1489.91	2853.29
Force	**2.60**	**6.75**	**75.28**	**2.94**	**14.21**	**85.85**	**13.59**	**155.15**	**324.17**
Cheetah	12.64	233.34	220.16	52.59	908.09	711.77	827.68	11012.47	8101.88

TABLE 9: Communication volume (MByte) of an ***inference*** pass with BatchSize $= 1$.

	CIFAR10			Tiny			ImageNet		
	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152
P-SecureML	65.93	381.39	1178.82	130.16	849.01	2082.17	1186.00	8361.98	15718.33
CryptGPU	2.32	53.59	236.17	13.32	214.12	677.61	226.08	2622.02	7376.14
P-Falcon	3.72	84.48	168.85	20.83	337.62	680.50	350.09	4134.47	8441.19
CrypTen	74.67	579.78	1409.07	178.98	1641.77	3034.04	2005.10	18069.92	27607.43
P-FantasticFour	7.01	159.50	300.42	39.24	637.43	1218.99	659.45	7805.96	15150.84
Force	**1.49**	**33.76**	**79.95**	**8.38**	**134.93**	**316.65**	**140.95**	**1652.33**	**3907.41**
Cheetah	40.10	951.35	773.51	249.24	3792.40	3091.30	4493.92	46450.00	37876.50

TABLE 10: Running time (Second) of a ***training*** pass in ***LAN*** setting with BatchSize $= 1$.

	CIFAR10			Tiny			ImageNet		
	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152
P-SecureML	1.62	4.55	29.20	7.53	5.99	27.81	7.41	28.82	65.51
CryptGPU	2.27	5.49	40.24	3.23	8.06	41.37	9.10	38.86	53.28
P-Falcon	0.75	2.44	12.08	0.96	3.04	13.55	4.13	16.14	35.78
CrypTen	13.48	40.86	27.68	18.39	50.34	33.35	FAIL	FAIL	74.07
P-FantasticFour	1.65	4.99	25.65	2.17	6.64	29.96	9.69	37.10	79.78
Force	**0.35**	**1.23**	**6.40**	**0.51**	**1.59**	**7.53**	**2.89**	**8.57**	**22.77**
PyTorch	0.0031	0.0067	0.0659	0.0027	0.0049	0.0637	0.0034	0.0077	0.0683

VGG16. Yet, CryptGPU is still 4.5 times slower than our Force.

When compared with the other three Piranha-based systems, P-SecureML, P-Falcon and P-FantasticFour, our Force is also faster in all experiments. On average, Force is 4.9 times faster than P-SecureML, 1.8 times faster than P-Falcon and 4 times faster than P-FantasticFour.

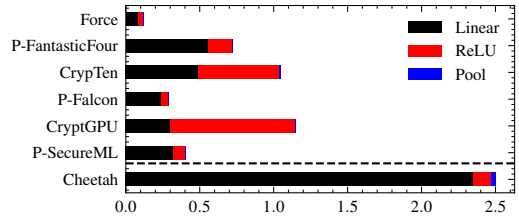
6.4. Linear vs. Non-Linear Operations

We Further look into how well $\mathcal{X}$-share accelerates common computation tasks. We are mainly interested in linear operations and non-linear operations (ReLU). Considering the existence of *Fused Batch Normalization*, which fuses batch normalization into convolution, we combine all convolution, matrix multiplication and batch normalization into one, as linear operations.

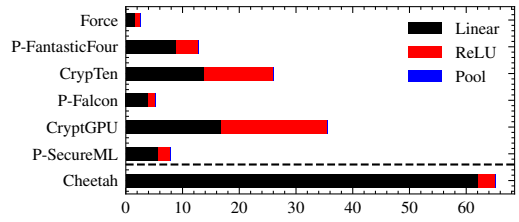
We plot the running time of different operations during an *inference* pass in *LAN* setting with BatchSize $= 1$ in Fig. 9. Due to the huge time difference between Cheetah and all other systems, all the experiments other than CIFAR10+AlexNet and CIFAR10+ResNet152 are rarely visible as bar charts. Thus we only include those two here.

We also make some micro-benchmark of matrix multiplication and ReLU in four Piranha-based systems. We perform $matmul$ and ReLU of different input data size and record the average running time. The results are plotted in Fig. 10. We omit some of the data of P-SecureML in Fig. 10a, since they are too large to show.

6.4.1. Linear Operations. As we can see from Fig. 9, Cheetah is extremely slow in linear operations. This principally results from the underlining cryptographic mechanism and the experiment platform. FHE requires much more

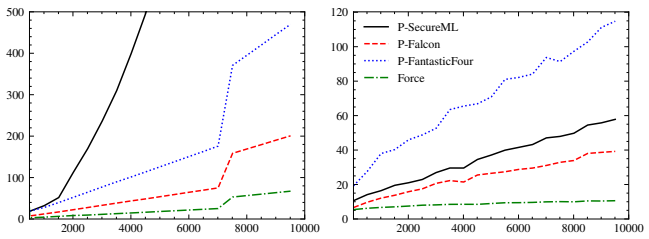


(a) CIFAR10 + AlexNet



(b) CIFAR10 + ResNet152

Fig. 9: Running time of different operations during an *inference* pass in *LAN* setting with BatchSize = 1. X-axis is time in seconds.



(a) Matrix Multiplication (b) ReLU

Fig. 10: Micro-benchmark of $matmul$ and ReLU in three Piranha-based systems. X-axis is data dimension and Y-axis is time in Milliseconds. For $matmul$, we multiply an $x \times x$ matrix by an $x \times 1$ vector.

computation than MPC. And different from CPUs, which are for generic computation, GPUs are specialized for data-intensive computations. It is not surprising that the CPU-FHE-based Cheetah is slower that all other GPU-MPC-based systems in linear operations.

Considering the implementation language difference, it is reasonable that Python-implemented CryptGPU and CrypTen are slower than the C++-implemented (P-SecureML, P-Falcon and Force). This is more observable in deeper networks, which involves more calculations, for example, in ResNet152 as showed in Fig. 9b.

Among those three C++-implemented systems, our Force completely outperforms the other two, with the optimizations described in §5.1 and §5.2. P-SecureML's multiplication does not scale well with problem size, as we can see from Fig. 10a. On the other side, both P-Falcon and Force scale linearly as the matrix becomes larger. Our Force is about 3 times faster than P-Falcon, which matches our analysis in §5.1.

6.4.2. Non-Linear Operations - ReLU. This is a different story. Originally, CPUs do better in non-linear operations than GPUs. When performing comparison in private inference, Cheetah is much faster than CryptGPU and CrypTen. Yet, still slower than other three Piranha-based systems.

ReLU accounts for more than 50% of total running time for both CryptGPU and CrypTen, which is quite surprising. In the plaintext situation, linear operations definitely dominate the whole procedure. We study their implementations and find out the reasons, as explained in §5.3.

P-SecureML and P-Falcon use similar ReLU implementations. The only difference is the underline sharing type and corresponding bit operations. Thus they have close performance, which can be also verified in Fig. 10b. Given our new sharing type $\mathcal{X}$-share and optimized relevant bit operations described in §5.3, Force is brought a huge boost. As we can see in Fig. 10b, the performance improves more as the problem size increases.

TABLE 11: Maximum batch size when training ImageNet in VGG16

Batch Size	1	2	4	8	16	32
P-SecureML	✓	✓	✓	✗	✗	✗
CryptGPU	✓	✗	✗	✗	✗	✗
P-Falcon	✓	✓	✓	✗	✗	✗
CrypTen	✗	✗	✗	✗	✗	✗
P-FantasticFour	✓	✓	✗	✗	✗	✗
Force	✓	✓	✓	✓	✓	✗

6.5. Communication Cost

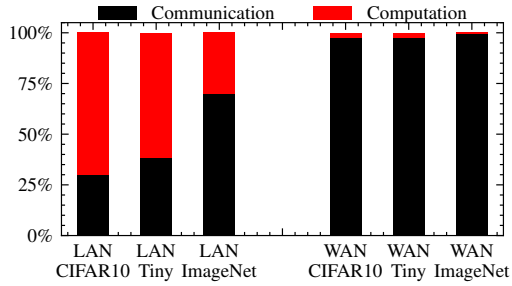


Fig. 11: Ratio of communication time and computation time when our Force performs ***inference*** on ResNet152 with BatchSize = 1.

Cheetah states that one of their main contribution is to reduce the communication volume and save communication time. We would like to point out that we do much better than them. More precisely, we have the minimal communication volume and lowest communication time in all datasets and models when performing inference with BatchSize = 1. We show the actual communication volume in Table 9.

However, we still notice high communication cost during all phases, especially for large datasets and *WAN* settings.

As an example, we plot the ratio of communication and computation time of our Force in Fig. 11. We can see that as the dataset gets larger, mainly the image dimensions, communication consumes more time. When the network latency is high, like in *WAN*, the whole running time is dominated by communication.

6.6. Memory Efficiency

Compared with RAM, which could easily reach $1TB$ nowadays, VRAM is an extremely limited resource. Typical GPU clusters are equipped with 16GB or maximum 24GB dedicated VRAM per card.

To measure the utilization efficiency of graphic memory, we run a simple experiment. We train one large dataset, `ImageNet`, on one of the large models, `VGG16`, with different batch sizes. Then we try to find out what the maximum batch size is for each system. The result is displayed in Table 11.

Our Force is the only system which supports training `ImageNet` on `VGG16` with BatchSize $= 8$ and BatchSize $= 16$. All the other systems can only train with batch size up to 4. CrypTen could not even train with BatchSize $= 1$.

6.7. Accuracy Comparison

TABLE 12: Accuracy comparison of Force's private inference protocol against `PyTorch`'s plaintext algorithm.

Inference	CIFAR10		Tiny		ImageNet	
	PyTorch	Force	PyTorch	Force	PyTorch	Force
AlexNet	69.65%	69.69%	26.38%	26.39%	22.84%	22.84%
VGG16	88.31%	88.34%	54.90%	54.89%	56.41%	56.42%
ResNet152	83.99%	83.98%	65.14%	65.15%	67.36%	67.36%

To measure the accuracy, we run both inference and training with Force. We first train the models on all the datasets with `PyTorch` to get pre-trained models. Starting from those pre-trained models, we perform the accuracy evaluation. We run all the evaluation with 26 bits of fixed-point precision, as suggested by Piranha.

For inference, we use the whole validation set of `CIFAR10` and a randomly selected subset of the validation sets of `Tiny` and `ImageNet`, so that the actual inference datasets contain 10,000 images each. The result is displayed in Table 12.

The experiment shows that models running over Force provide almost the same accuracy as the plaintext models, only with a tiny relative error of less than 0.1% for all models and datasets.

For training, we use the whole training set of `CIFAR10`, as well as the whole validation set for validation. Starting from a pre-trained model, we train `AlexNet` on `CIFAR10` with both Force and `PyTorch` for 9 epochs. We plot the validation accuracy in Fig. 12.

After 9 epochs, the accuracy of `PyTorch` is 49.59%, of Force is 49.71%, which is even 0.12% higher. We also plot the validation accuracy of Piranha in Fig. 12, which indicates that Force has no accuracy loss comparing with Piranha. Note that the three Piranha-based systems (P-SecureML, P-Falcon and P-FantasticFour) have the same accuracy, so we only plot a single line for them.

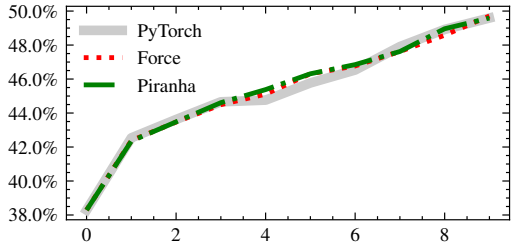


Fig. 12: Validation accuracy of 9 training epochs when training `AlexNet` on `CIFAR10`.

7. Conclusion

In this paper, we construct a powerful system Force for PPML. Our implementation and thorough evaluation showcase that Force is by far the most efficient in terms of time, memory consumption and overall performance. It can be meaningful future work to extend Force with security against fully malicious adversaries, guarantee of delivery, and generalization for any even number of parties.

References

[1] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 805–817.

[2] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More efficient oblivious transfer extensions with security for malicious adversaries," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 673–701.

[3] Y. Aumann and Y. Lindell, "Security against covert adversaries: Efficient protocols for realistic adversaries," *Journal of Cryptology*, vol. 23, no. 2, pp. 281–343, 2010.

[4] J. Bar-Ilan and D. Beaver, "Non-cryptographic fault-tolerant computing in constant number of rounds of interaction," in *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, 1989, pp. 201–209.

[5] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 351–371.

[6] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving com-

putations," in *European Symposium on Research in Computer Security*. Springer, 2008, pp. 192–206.

[7] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson, "High-performance secure multi-party computation for data mining applications," *International Journal of Information Security*, vol. 11, no. 6, pp. 403–418, 2012.

[8] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for federated learning on user-held data," *arXiv preprint arXiv:1611.04482*, 2016.

[9] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "Flash: fast and robust framework for privacy-preserving machine learning," *Cryptology ePrint Archive*, 2019.

[10] J. Cabrero-Holgueras and S. Pastrana, "Sok: Privacy-preserving computation techniques for deep learning," *Proceedings on Privacy Enhancing Technologies*, vol. 2021, no. 4, pp. 139–162, 2021.

[11] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[12] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai, "Universally composable two-party and multi-party secure computation," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, 2002, pp. 494–503.

[13] O. Catrina and S. d. Hoogh, "Improved primitives for secure multiparty integer computation," in *International Conference on Security and Cryptography for Networks*. Springer, 2010, pp. 182–199.

[14] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "Astra: high throughput 3pc over rings with application to secure prediction," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019, pp. 81–92.

[15] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4pc framework for privacy preserving machine learning," *arXiv preprint arXiv:1912.02631*, 2019.

[16] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[17] A. Dalskov, D. Escudero, and M. Keller, "Fantastic four:{Honest-Majority}{Four-Party} secure computation with malicious security," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2183–2200.

[18] I. Damgård, C. Orlandi, and M. Simkin, "Black-box transformations from passive to covert security with public verifiability," in *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II 40*. Springer, 2020, pp. 647–676.

[19] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Annual Cryptology Conference*. Springer, 2012, pp. 643–662.

[20] D. Demmler, T. Schneider, and M. Zohner, "Aby-a framework for efficient mixed-protocol secure two-party computation." in *NDSS*, 2015.

[21] C. Dwork, A. Roth *et al.*, "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014.

[22] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for mpc over mixed arithmetic-binary circuits," in *Annual International Cryptology conference*. Springer, 2020, pp. 823–852.

[23] D. Evans, V. Kolesnikov, M. Rosulek *et al.*, "A pragmatic introduction to secure multi-party computation," *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2-3, pp. 70–246, 2018.

[24] T. K. Frederiksen and J. B. Nielsen, "Fast and maliciously secure two-party computation using the gpu," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 339–356.

[25] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-throughput secure three-party computation for malicious adversaries and an honest majority," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2017, pp. 225–255.

[26] D. Genkin, Y. Ishai, and A. Polychroniadou, "Efficient multi-party computation: from passive to active security via secure simd circuits," in *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II 35*. Springer, 2015, pp. 721–741.

[27] O. GOLDREICH, "How to play any mental game," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing, STOC'87, New York, NY, USA*. ACM, 1987, pp. 218–229.

[28] Google LLC, "Tensorflow privacy," 2019. [Online]. Available: https://github.com/tensorflow/privacy

[29] D. Harris, "A taxonomy of parallel prefix networks," in *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, vol. 2. IEEE, 2003, pp. 2213–2217.

[30] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "Sok: General purpose compilers for secure multi-party computation," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 1220–1237.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[32] E. Hesamifard, H. Takabi, M. Ghasemi, and R. N. Wright, "Privacy-preserving machine learning as a service." *Proc. Priv. Enhancing Technol.*, vol. 2018, no. 3, pp. 123–142, 2018.

[33] C. Hong, J. Katz, V. Kolesnikov, W.-j. Lu, and

X. Wang, "Covert security with public verifiability: Faster, leaner, and simpler," in *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part III 38*. Springer, 2019, pp. 97–121.

[34] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure {Two-Party} deep neural network inference," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 809–826.

[35] N. Husted, S. Myers, A. Shelat, and P. Grubbs, "Gpu and cpu parallelization of honest-but-curious secure two-party computation," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 169–178.

[36] D. Ikarashi, R. Kikuchi, K. Hamada, and K. Chida, "Actively private and correct mpc scheme in $t < n/2$ from passively secure schemes with small overhead," *Cryptology ePrint Archive*, 2014.

[37] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[38] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "{GAZELLE}: A low latency framework for secure neural network inference," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1651–1669.

[39] R. Kanagavelu, Z. Li, J. Samsudin, Y. Yang, F. Yang, R. S. M. Goh, M. Cheah, P. Wiwatphonthana, K. Akkarajitsakul, and S. Wang, "Two-phase multi-party computation enabled privacy-preserving federated learning," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 410–419.

[40] J. Katz, S. Ranellucci, M. Rosulek, and X. Wang, "Optimizing authenticated garbling for faster secure two-party computation," 2018, https://ia.cr/2018/578.

[41] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: making spdz great again," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 158–189.

[42] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "Crypten: Secure multi-party computation meets machine learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 4961–4973, 2021.

[43] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: super-fast and robust privacy-preserving machine learning," *CoRR*, 2020. [Online]. Available: https://arxiv.org/abs/2005.10296

[44] N. Koti, A. Patra, R. Rachuri, and A. Suresh, "Tetrad: Actively secure 4pc for secure training and inference," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[45] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012.

[47] KU Leuven - COSIC, "SCALE Multiparty AlgorithMs Basic Argot : MAMBA," 2022. [Online]. Available: https://github.com/KULeuven-COSIC/SCALE-MAMBA

[48] Y. Le and X. Yang, "Tiny imagenet visual recognition challenge," *CS 231N*, vol. 7, no. 7, p. 3, 2015.

[49] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, "Muse: Secure inference resilient to malicious clients," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2201–2218.

[50] Y. Lindell, "How to simulate it–a tutorial on the simulation proof technique," *Tutorials on the Foundations of Cryptography*, pp. 277–346, 2017.

[51] Y. Lindell and B. Pinkas, "An efficient protocol for secure two-party computation in the presence of malicious adversaries," 2008, https://ia.cr/2008/049.

[52] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 619–631.

[53] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2505–2522.

[54] P. Mohassel and P. Rindal, "Aby3: A mixed protocol framework for machine learning," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 35–52.

[55] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *2017 IEEE symposium on security and privacy (SP)*. IEEE, 2017, pp. 19–38.

[56] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra, "A new approach to practical active-secure two-party computation," in *Annual Cryptology Conference*. Springer, 2012, pp. 681–700.

[57] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman, "Sok: Security and privacy in machine learning," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 399–414.

[58] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[59] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "{ABY2. 0}: Improved {Mixed-Protocol} secure {Two-Party} computation," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2165–

2182.
[60] A. Patra and A. Suresh, "Blaze: blazing fast privacy-preserving machine learning," *arXiv preprint arXiv:2005.09042*, 2020.
[61] S. Pu, P. Duan, and J.-C. Liu, "Fastplay-a parallelization model and implementation of smc on cuda based gpu cluster architecture," *Cryptology ePrint Archive*, 2011.
[62] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 325–342.
[63] D. Rotaru and T. Wood, "Marbled circuits: Mixing arithmetic and boolean circuits with active security," in *International Conference on Cryptology in India*. Springer, 2019, pp. 227–249.
[64] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.
[65] P. Scholl, M. Simkin, and L. Siniscalchi, "Multiparty computation with covert security and public verifiability," *Cryptology ePrint Archive*, 2021.
[66] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[67] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "Cryptgpu: Fast privacy-preserving machine learning on the gpu," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1021–1038.
[68] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou, "A hybrid approach to privacy-preserving federated learning," in *Proceedings of the 12th ACM workshop on artificial intelligence and security*, 2019, pp. 1–11.
[69] United Nations Conference on Trade and Development, "Data protection and privacy legislation worldwide," 2022. [Online]. Available: https://unctad.org/page/data-protection-and-privacy-legislation-worldwide
[70] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training." *Proc. Priv. Enhancing Technol.*, vol. 2019, no. 3, pp. 26–49, 2019.
[71] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "FALCON: honest-majority maliciously secure framework for private deep learning," *CoRR*, 2020. [Online]. Available: https://arxiv.org/abs/2004.02229
[72] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," 2017, https://ia.cr/2017/030.
[73] J.-L. Watson, S. Wagh, and R. A. Popa, "Piranha: A {GPU} platform for secure computation," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 827–844.
[74] A. Wigderson, M. Or, and S. Goldwasser, "Completeness theorems for noncryptographic fault-tolerant distributed computations," in *Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC'88)*, 1988, pp. 1–10.
[75] Q. Zhang, Y. Zhao, L. Li, J. Zhang, Q. Zhang, Y. Zhou, D. Yin, S. Tan, and S. Yin, "Morse-stf: A privacy preserving computation system," *arXiv preprint arXiv:2109.11726*, 2021.

Appendix A. Ideal Functionalities in 4PC

See Figure 13, 14 and 15.

Appendix B. CNN, an Example

See Fig. 16.

Appendix C. Security Proof

C.1. Multiplication Protocol Π_{Mult}

Theorem 1. *Protocol Π_{Mult} UC-realizes $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{MULT}}$ in the $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$-hybrid model, in the presence of a semi-honest adversary who can corrupt $\mathbf{P_i}$ where $\mathbf{P_i} \in \{\mathbf{P_A}, \mathbf{P_B}, \mathbf{P_C}, \mathbf{P_D}\}$, with static corruption.*

Proof Sketch: Suppose parties hold $[x]_{\mathsf{AC}}$, $[y]_{\mathsf{AB}}$ and are willing to compute $[z]_{\mathsf{AC}}$. We construct an adversary $\mathcal{S}$ interacting $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{MULT}}$ such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol Π_{Mult} in the real world or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{MULT}}$. Since each $\mathbf{P_i}$ is corrupted by a semi-honest $\mathcal{A}$, even if $\mathcal{A}$ is able to modify the input tape of $\mathbf{P_i}$, this is actually the modified value sent from environment machine $\mathcal{Z}$ [12]. Suppose $\mathbf{P_A}$ is corrupted, the secret value x and y are AC and AB shared, and the output z is AC shared. Recall that $\mathcal{S}$ has already received the input of $\mathcal{A}$, denoted as $[x]_{\mathsf{AC}}^{\mathbf{P_A}}$, $[y]_{\mathsf{AB}}^{\mathbf{P_A}}$, which is modified by $\mathcal{Z}$. First, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$, sends k_{Zero} and k'_{Zero} to $\mathcal{A}$ as PRF keys computing *zero sharing* (other keys are irrelevant in this case). Note that $\mathcal{S}$ receives $\mathcal{A}$'s output by sending its input to $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{MULT}}$, denoted as $[z]_{\mathsf{AC}}^{\mathbf{P_A}}$. Thus, to perfectly simulate $\mathbf{P_B}$, $\mathcal{S}$ firstly computes $r^{\mathbf{P_A}} = \mathsf{PRF}_{k_{\mathsf{Zero}}}(\mathsf{sid}) - \mathsf{PRF}_{k'_{\mathsf{Zero}}}(\mathsf{sid})$ just as $\mathcal{A}$ will do and sends $r^{\mathbf{P_A}}$ to $\mathcal{A}$. Then $\mathcal{S}$ computes and sends $m^{\mathbf{P_B}} = [z]_{\mathsf{AC}}^{\mathbf{P_A}} - r^{\mathbf{P_A}} - [x]_{\mathsf{AC}}^{\mathbf{P_A}}[y]_{\mathsf{AB}}^{\mathbf{P_A}}$ to $\mathcal{A}$. Note that in the real protocol execution, $\mathbf{P_B}$ computes $[x]_{\mathsf{AC}}^{\mathbf{P_B}}[y]_{\mathsf{AB}}^{\mathbf{P_B}} + r^{\mathbf{P_B}}$, where $r^{\mathbf{P_B}}$ is distributed uniformly at random to the environment machine $\mathcal{Z}$. Thus, the message $m^{\mathbf{P_B}}$ sent from $\mathcal{S}$ in the ideal execution is indistinguishable from $[x]_{\mathsf{AC}}^{\mathbf{P_B}}[y]_{\mathsf{AB}}^{\mathbf{P_B}} + r^{\mathbf{P_B}}$ computed by $\mathbf{P_B}$ in the real protocol execution. This yields a perfect simulation.

$\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$

Private inputs: No.
Public inputs: Public parameters.
Output: Triples to support other computation.

- Upon receiving a corrupt query from $\mathbf{P_i}$
  - Send k_{Zero} and k'_{Zero} to $\mathbf{P_i}$.
  - Send $(k_{\mathsf{ACtoAB}}, k'_{\mathsf{ACtoAB}})$, $(k_{\mathsf{ABtoAC}}, k'_{\mathsf{ABtoAC}})$ to $\mathbf{P_i}$.
- Upon receiving $(\mathsf{ZeroSGen}, \mathbf{P_i}, \mathsf{sid})$ from the corrupted $\mathbf{P_i}$, and other $\mathbf{P_j} \in \mathcal{P}/\{\mathbf{P_i}\}$:
  - $r^{\mathbf{P_i}} := \mathsf{PRF}_{k_{\mathsf{Zero}}}(\mathsf{sid}) - \mathsf{PRF}_{k'_{\mathsf{Zero}}}(\mathsf{sid})$.
  - Sample $r^{\mathbf{P_j}} \xleftarrow{\$} \mathbb{Z}_{2^\ell}$, where $\mathbf{P_j} \in \mathcal{P}\backslash\{\mathbf{P_i}\}$ and $\sum r^{\mathbf{P_j}} = 0$ for all $\mathbf{P_j} \in \mathcal{P}$.
  - Send $r^{\mathbf{P_j}}$ to $\mathbf{P_j}$ for all $\mathbf{P_j} \in \mathcal{P}$.
- Upon receiving $(\mathsf{CMSGen}, \psi\mathsf{to}\phi, \mathbf{P_i}, \mathsf{sid})$ from the corrupted party $\mathbf{P_i}$ and other $\mathbf{P_j}$ where $\mathbf{P_j} \in \mathcal{P}/\{\mathbf{P_i}\}$:
  - If $\psi\mathsf{to}\phi = \mathsf{ACtoAB}$:
    - If $\mathbf{P_i} \in \{\mathbf{P_B}, \mathbf{P_D}\}$, compute $r := \mathsf{PRF}_{k_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k'_{\mathsf{ACtoAB}}}(\mathsf{sid})$, set $[\![r]\!]^{\mathbf{P_B}}_{\mathsf{ACtoAB}} = [\![r]\!]^{\mathbf{P_D}}_{\mathsf{ACtoAB}} := r$. Choose $[\![r]\!]^{\mathbf{P_A}}_{\mathsf{ACtoAB}} \xleftarrow{\$} \mathbb{Z}_{2^\ell}$, $[\![r]\!]^{\mathbf{P_C}}_{\mathsf{ACtoAB}} \xleftarrow{\$} \mathbb{Z}_{2^\ell}$, where $[\![r]\!]^{\mathbf{P_A}}_{\mathsf{ACtoAB}} + [\![r]\!]^{\mathbf{P_C}}_{\mathsf{ACtoAB}} = [\![r]\!]^{\mathbf{P_B}}_{\mathsf{ACtoAB}} = [\![r]\!]^{\mathbf{P_D}}_{\mathsf{ACtoAB}}$.
    - If $\mathbf{P_i} \in \{\mathbf{P_A}, \mathbf{P_C}\}$, compute $r_i := \mathsf{PRF}_{k_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k'_{\mathsf{ACtoAB}}}(\mathsf{sid})$, set $[\![r]\!]^{\mathbf{P_i}}_{\mathsf{ACtoAB}} := r_i$. Choose $[\![r]\!]^{\mathbf{P_h}}_{\mathsf{ACtoAB}} \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ where $\mathbf{P_h} \in \{\mathbf{P_A}, \mathbf{P_C}\}/\mathbf{P_i}$. Compute $r := [\![r]\!]^{\mathbf{P_A}}_{\mathsf{ACtoAB}} + [\![r]\!]^{\mathbf{P_C}}_{\mathsf{ACtoAB}}$, set $[\![r]\!]^{\mathbf{P_B}}_{\mathsf{ACtoAB}} = [\![r]\!]^{\mathbf{P_D}}_{\mathsf{ACtoAB}} := r$.
  - If $\psi\mathsf{to}\phi = \mathsf{ABtoAC}$, switch the role of $\mathbf{P_B}$ and $\mathbf{P_C}$, use $(k_{\mathsf{ABtoAC}}, k'_{\mathsf{ABtoAC}})$ instead of $(k_{\mathsf{ACtoAB}}, k'_{\mathsf{ACtoAB}})$ and do the same as above.
  - Send $[\![r]\!]^{\mathbf{P_j}}_{\psi\mathsf{to}\phi}$ to $\mathbf{P_j}$ for all $\mathbf{P_j} \in \mathcal{P}$.
- Upon receiving $(\mathcal{X}\text{-}\mathsf{dabitGen}, \psi, \phi, \ell^{\mathsf{B}}, \mathbf{P_i}, \mathsf{sid})$ from parties where $\mathbf{P_i} \in \mathcal{P}$:
  - Sample $(\alpha_0, \alpha_1) \xleftarrow{\$} \mathbb{Z}_{2^\ell}$.
  - Sample $\{(\beta_0^0, \beta_0^1), (\beta_1^0, \beta_1^1), \cdots, (\beta_{\ell^{\mathsf{B}}-1}^0, \beta_{\ell^{\mathsf{B}}-1}^1)\} \xleftarrow{\$} \mathbb{Z}_2$, such that $b := \alpha_0 + \alpha_1$ and $\langle b_i \rangle := \oplus_{j=0}^{1} \beta_i^j$.
  - If $\psi = \mathsf{AC}$, send α_0 to $\mathbf{P_A}$ and $\mathbf{P_B}$, α_1 to $\mathbf{P_C}$ and $\mathbf{P_D}$. Otherwise, send α_0 to $\mathbf{P_A}$ and $\mathbf{P_C}$, α_1 to $\mathbf{P_B}$ and $\mathbf{P_D}$.
  - If $\phi = \mathsf{AC}$, send $\{\beta_i^0\}_{i\in[\ell^{\mathsf{B}}-1]}$ to $\mathbf{P_A}$ and $\mathbf{P_B}$, $\{\beta_i^1\}_{i\in[\ell^{\mathsf{B}}-1]}$ to $\mathbf{P_C}$ and $\mathbf{P_D}$. Otherwise, send $\{\beta_i^0\}_{i\in[\ell^{\mathsf{B}}-1]}$ to $\mathbf{P_A}$ and $\mathbf{P_C}$, $\{\beta_i^1\}_{i\in[\ell^{\mathsf{B}}-1]}$ to $\mathbf{P_B}$ and $\mathbf{P_D}$.

Fig. 13: Four Party Functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$.

C.2. Change Share Mode Protocol Π_{chMode}

Theorem 2. *Protocol Π_{chMode} UC-realizes $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMode}}$ in the $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$-hybrid model, in the presence of a semi-honest adversary who can corrupt $\mathbf{P_i}$ where $\mathbf{P_i} \in \{\mathbf{P_A}, \mathbf{P_B}, \mathbf{P_C}, \mathbf{P_D}\}$, with static corruption.*

$\mathcal{F}_{\mathsf{4PC}}^{\mathsf{MULT}}$

Private inputs: Let $\mathcal{P} = \{\mathbf{P_A}, \mathbf{P_B}, \mathbf{P_C}, \mathbf{P_D}\}$. Each party $\mathbf{P_i} \in \mathcal{P}$ holds $[x]_\psi^{\mathbf{P_i}}$ and $[y]_\phi^{\mathbf{P_i}}$, where $\psi \neq \phi$.
Public inputs: Public parameters, θ.
Outputs: $[z]_\theta^{\mathbf{P_i}}$ for all $\mathbf{P_i} \in \mathcal{P}$, with $z = x \cdot y$.
Initialization: All $\mathcal{L}_{\mathsf{sid}} = \emptyset$.
Compute:

- Upon receiving $(\mathsf{MULT}, [x]_\psi^{\mathbf{P_i}}, [y]_\phi^{\mathbf{P_i}}, \mathbf{P_i}, \mathsf{sid})$ from any party $\mathbf{P_i} \in \mathcal{P}$.
- If $\mathcal{L}_{\mathsf{sid}} = \perp$, do nothing.
- Check if every other record $(\mathsf{MULT}, [x]_\psi^{\mathbf{P_j}}, [y]_\phi^{\mathbf{P_j}}, \mathbf{P_j}, \mathsf{sid})$, $\mathbf{P_j} \in \mathcal{P}\backslash\{\mathbf{P_i}\}$ is in $\mathcal{L}_{\mathsf{sid}}$:
  - If yes, compute $z = \mathsf{rec}([x]_\psi)\,\mathsf{rec}([y]_\phi)$, where $\mathsf{rec}()$ is the reconstruction function.
    Choose $r \xleftarrow{\$} \mathbb{Z}_{2^\ell}$. If $\theta = \mathsf{AC}$, set $[z]_\theta^{\mathbf{P_A}} = [z]_\theta^{\mathbf{P_B}} := z - r$ and $[z]_\theta^{\mathbf{P_B}} = [z]_\theta^{\mathbf{P_D}} := r$. Otherwise $\theta = \mathsf{AB}$, set $[z]_\theta^{\mathbf{P_A}} = [z]_\theta^{\mathbf{P_C}} := z - r$ and $[z]_\theta^{\mathbf{P_C}} = [z]_\theta^{\mathbf{P_D}} := r$.
    Send $(\mathbf{P_j}, \mathsf{sid}, [z]_\theta^{\mathbf{P_j}})$ to $\mathbf{P_j}$ for all $\mathbf{P_j} \in \mathcal{P}$.
    Set $\mathcal{L}_{\mathsf{sid}} = \perp$.
  - Otherwise, set $\mathcal{L}_{\mathsf{sid}} \leftarrow \mathcal{L}_{\mathsf{sid}} \cup \{(\mathsf{MULT}, [x]_\psi^{\mathbf{P_i}}, [y]_\phi^{\mathbf{P_i}}, \mathbf{P_i}, \mathsf{sid})\}$.

Fig. 14: Four Party Functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{MULT}}$.

$\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Comp}}$

Private inputs: Let $\mathcal{P} = \{\mathbf{P_A}, \mathbf{P_B}, \mathbf{P_C}, \mathbf{P_D}\}$. Each party $\mathbf{P_i} \in \mathcal{P}$ holds $[x]_\psi^{\mathbf{P_i}}$.
Public inputs: Public parameters.
Outputs: $\langle b \rangle_\psi^{\mathbf{P_i}}$ for all $\mathbf{P_i} \in \mathcal{P}$, with $b = 1$ if $x \geq 0$ and $b = 0$ otherwise.
Compute:

- Upon receiving $(\mathsf{Comp}, [x]_\psi^{\mathbf{P_i}}, \mathbf{P_i}, \mathsf{sid})$ from any party $\mathbf{P_i} \in \mathcal{P}$.
- If $\mathcal{L}_{\mathsf{sid}} = \perp$, do nothing.
- Check if every other record $(\mathsf{Comp}, [x]_\psi^{\mathbf{P_j}}, \mathbf{P_j}, \mathsf{sid})$, $\mathbf{P_j} \in \mathcal{P}\backslash\{\mathbf{P_i}\}$ is in $\mathcal{L}_{\mathsf{sid}}$:
  - If yes, compute $x = \mathsf{rec}([x]_\psi^{\mathbf{P_i}})$, where $\mathsf{rec}()$ is the reconstruction function.
    If $x \geq 0$, set $b = 1$, otherwise set $b = 0$.
    Choose $r \xleftarrow{\$} \mathbb{Z}_2$, if $\psi = \mathsf{AC}$, set $\langle b \rangle_\psi^{\mathbf{P_A}} = \langle b \rangle_\psi^{\mathbf{P_B}} := b \oplus r$ and $\langle b \rangle_\psi^{\mathbf{P_C}} = \langle b \rangle_\psi^{\mathbf{P_D}} := r$. Otherwise if $\psi = \mathsf{AB}$, set $\langle b \rangle_\psi^{\mathbf{P_A}} = \langle b \rangle_\psi^{\mathbf{P_C}} := b \oplus r$ and $\langle b \rangle_\psi^{\mathbf{P_B}} = \langle b \rangle_\psi^{\mathbf{P_D}} := r$.
    Send $(\mathbf{P_j}, \mathsf{sid}, \langle b \rangle_\psi^{\mathbf{P_j}})$ to $\mathbf{P_j}$ for all $\mathbf{P_j} \in \mathcal{P}$.
    Set $\mathcal{L}_{\mathsf{sid}} = \perp$.
  - Otherwise, set $\mathcal{L}_{\mathsf{sid}} \leftarrow \mathcal{L}_{\mathsf{sid}} \cup \{(\mathsf{Comp}, [x]_\psi^{\mathbf{P_i}}, \mathbf{P_i}, \mathsf{sid})\}$.

Fig. 15: Four Party Functionality for Comparison $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Comp}}$.

Proof Sketch: Suppose parties hold $[x]_{\mathsf{AC}}$ are willing to compute $[x]_{\mathsf{AB}}$. We construct an adversary $\mathcal{S}$ interacting $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMode}}$ such that no environment $\mathcal{Z}$ can tell with non-

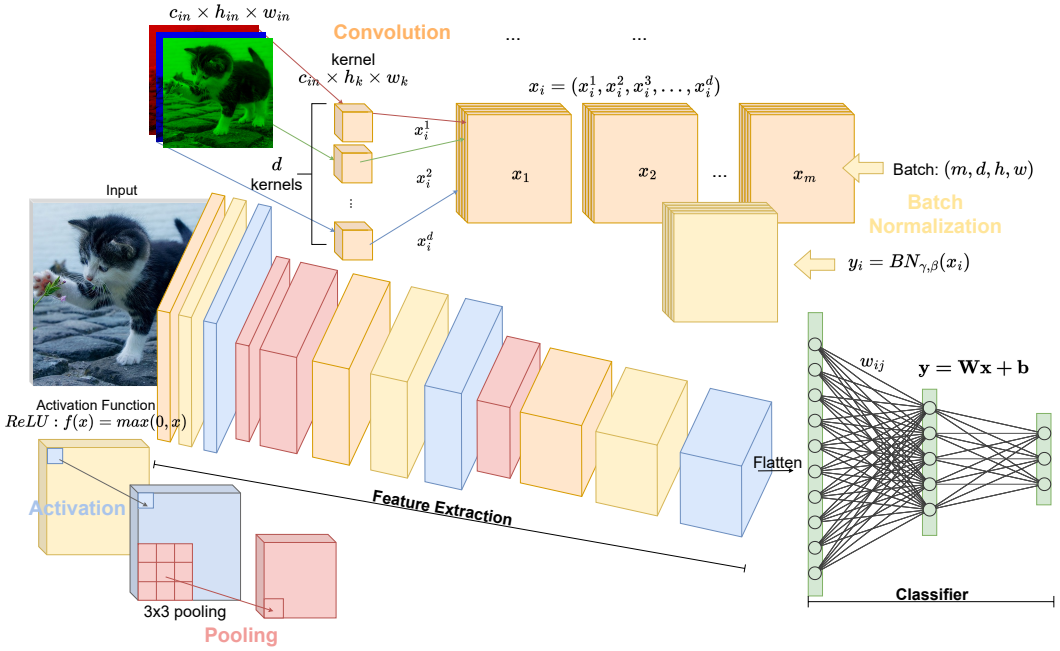


Fig. 16: An example of CNN pipeline and corresponding layer decomposition.

negligible probability whether it is interacting with $\mathcal{A}$ and the protocol Π_{Mult} in the real world or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMode}}$. Suppose $\mathbf{P_A}$ is corrupted, the $\mathcal{A}$'s input is denoted as $[x]_{\mathsf{AC}}^{\mathbf{P_A}}$, and the output should be the same secret value x in $[\cdot]_{\mathsf{AB}}$-sharing. During preprocessing stage, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$, which sends k_{ACtoAB} and k'_{ACtoAB} to $\mathcal{A}$ (other keys are irrelevant in this case). Note that $\mathcal{S}$ receives $\mathcal{A}$'s output by sending its input to $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMode}}$, denoted as $[x]_{\mathsf{AB}}^{\mathbf{P_A}}$. To perfectly simulate $\mathbf{P_C}$, $\mathcal{S}$ firstly computes $[\![r]\!]_{\mathsf{ACtoAB}}^{\mathbf{P_A}} = \mathsf{PRF}_{k_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k'_{\mathsf{ACtoAB}}}(\mathsf{sid})$ just as $\mathcal{A}$ will do and sends $[\![r]\!]_{\mathsf{ACtoAB}}^{\mathbf{P_A}}$ to $\mathcal{A}$. Then acting as an honest $\mathbf{P_C}$, $\mathcal{S}$ computes and sends $m^{\mathbf{P_C}} = [x]_{\mathsf{AB}}^{\mathbf{P_A}} - [x]_{\mathsf{AC}}^{\mathbf{P_A}} - [\![r]\!]_{\mathsf{ACtoAB}}^{\mathbf{P_A}}$ to $\mathbf{P_A}$. Remark that in the real protocol execution, $\mathbf{P_C}$ computes $[x]_{\mathsf{AC}}^{\mathbf{P_C}} - [\![r]\!]_{\mathsf{ACtoAB}}^{\mathbf{P_C}}$, where $[\![r]\!]_{\mathsf{ACtoAB}}^{\mathbf{P_C}}$ is distributed uniformly at random to the environment machine $\mathcal{Z}$. Thus, the message $m^{\mathbf{P_C}}$ sent by $\mathcal{S}$ in the ideal execution is indistinguishable from $[x]_{\mathsf{AC}}^{\mathbf{P_C}} - [\![r]\!]_{\mathsf{ACtoAB}}^{\mathbf{P_C}}$ computed by $\mathbf{P_C}$ in the real protocol execution. This yields a perfect simulation. We now consider the case when $\mathbf{P_B}$ is corrupted, the $\mathcal{A}$'s input is denoted as $[x]_{\mathsf{AC}}^{\mathbf{P_B}}$, and the output should be the same secret value x in $[\cdot]_{\mathsf{AB}}$-sharing. Again, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$ and sends k_1 and k_2 to $\mathcal{A}$. $\mathcal{S}$ computes $[\![r]\!]_{\mathsf{ACtoAB}}^{\mathbf{P_B}} = \mathsf{PRF}_{k_{\mathsf{ACtoAB}}}(\mathsf{sid}) - \mathsf{PRF}_{k'_{\mathsf{ACtoAB}}}(\mathsf{sid})$ just as $\mathcal{A}$ will do and sends $[\![r]\!]_{\mathsf{ACtoAB}}^{\mathbf{P_B}}$ to $\mathcal{A}$. Then, $\mathcal{S}$ sends $c^{\mathbf{P_B}} = [\![r]\!]_{\mathsf{ACtoAB}}^{\mathbf{P_B}}$ to $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMode}}$ and halts. Remark that the simulation in this case is a local simulation, since $\mathcal{A}$ just locally outputs $[x]_{\mathsf{AB}}^{\mathbf{P_B}}$. Thus, this simulation is perfect.

C.3. Bit to Arithmetic Protocol Π_{BitToA}

Theorem 3. *Protocol Π_{BitToA} UC-realizes $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{B2A}}$ in the $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$-hybrid model, in the presence of a semi-honest adversary who can corrupt $\mathbf{P_i}$ where $\mathbf{P_i} \in \{\mathbf{P_A}, \mathbf{P_B}, \mathbf{P_C}, \mathbf{P_D}\}$, with static corruption.*

Proof Sketch: Suppose parties hold $\langle x \rangle_{\mathsf{AC}}$ with $x \in \mathbb{Z}_2$ and are willing to compute $[x]_{\mathsf{AB}}$. We construct an adversary $\mathcal{S}$ interacting $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{B2A}}$ such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol Π_{BitToA} in the real world or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{B2A}}$. Suppose $\mathbf{P_A}$ is corrupted, the $\mathcal{A}$'s input is denoted as $\langle x \rangle_{\mathsf{AC}}^{\mathbf{P_A}}$ and the output should be AB shared. During preprocessing stage, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$, which sends $[b]_{\mathsf{AB}}^{\mathbf{P_A}}$, $\langle b \rangle_{\mathsf{AC}}^{\mathbf{P_A}}$ to $\mathcal{A}$. Upon receiving $m^{\mathbf{P_A}} = \langle x \rangle_{\mathsf{AC}}^{\mathbf{P_A}} \oplus \langle b \rangle_{\mathsf{AC}}^{\mathbf{P_A}}$ from $\mathbf{P_A}$, $\mathcal{S}$ chooses $m^{\mathbf{P_C}} \xleftarrow{\$} \mathbb{Z}_2$, then sends to $\mathbf{P_A}$. If $m^{\mathbf{P_A}} \oplus m^{\mathbf{P_C}} = 0$, $\mathcal{S}$ sends $[b]_{\mathsf{AB}}^{\mathbf{P_A}}$ to $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{B2A}}$, otherwise $1 - [b]_{\mathsf{AB}}^{\mathbf{P_A}}$. We notice that in the real protocol execution, $\mathbf{P_C}$ computes $\langle x \rangle_{\mathsf{AC}}^{\mathbf{P_C}} \oplus \langle b \rangle_{\mathsf{AC}}^{\mathbf{P_C}}$, where $\langle b \rangle_{\mathsf{AC}}^{\mathbf{P_C}}$ is distributed uniformly at random to the environment machine $\mathcal{Z}$. Thus, the ideal execution and the real protocol execution are indistinguishable. The simulation in this case is perfect.

$\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMode}}$

Private inputs: Let $\mathcal{P} = \{\mathbf{P_A}, \mathbf{P_B}, \mathbf{P_C}, \mathbf{P_D}\}$. Each party $\mathbf{P_i} \in \mathcal{P}$ holds $[x]_{\psi}^{\mathbf{P_i}}$.
Public inputs: Public parameters.
Outputs: $[x]_{\phi}^{\mathbf{P_i}}$ for all $\mathbf{P_i} \in \mathcal{P}$, with $\phi \neq \psi$.
Initialization: All $\mathcal{L}_{\mathsf{sid}} = \emptyset$.
Compute:

- Upon receiving $(\mathsf{chMode}, [x]_{\psi}^{\mathbf{P_i}}, \mathbf{P_i}, \mathsf{sid})$ from the corrupted $\mathbf{P_i}$ where $\mathbf{P_i} \in \mathcal{P}$.
- If $\mathcal{L}_{\mathsf{sid}} = \perp$, do nothing.
- Check if every other record $(\mathsf{chMode}, [x]_{\psi}^{\mathbf{P_j}}, \mathbf{P_j}, \mathsf{sid})$, $\mathbf{P_j} \in \mathcal{P} \backslash \{\mathbf{P_i}\}$ is in $\mathcal{L}_{\mathsf{sid}}$:
  - If yes, compute $x = \mathsf{rec}([x]_{\psi})$, where $\mathsf{rec}()$ is the reconstruction function.
    If $\phi = \mathsf{AB}$:
    * If $\mathbf{P_i} \in \{\mathbf{P_B}, \mathbf{P_D}\}$, upon receiving $c^{\mathbf{P_i}}$ from $\mathbf{P_i}$, set $[x]_{\phi}^{\mathbf{P_B}} = [x]_{\phi}^{\mathbf{P_D}} := c^{\mathbf{P_i}}$. Set $[x]_{\phi}^{\mathbf{P_A}} = [x]_{\phi}^{\mathbf{P_C}} := x - c^{\mathbf{P_i}}$.
    * If $\mathbf{P_i} \in \{\mathbf{P_A}, \mathbf{P_C}\}$, choose $r \xleftarrow{\$} Z_{2^\ell}$, set $[x]_{\phi}^{\mathbf{P_A}} = [x]_{\phi}^{\mathbf{P_C}} := r$. Set $[x]_{\phi}^{\mathbf{P_B}} = [x]_{\phi}^{\mathbf{P_D}} := x - r$.

    If $\phi = \mathsf{AC}$:
    * If $\mathbf{P_i} \in \{\mathbf{P_C}, \mathbf{P_D}\}$, upon receiving $c^{\mathbf{P_i}}$ from $\mathbf{P_i}$, set $[x]_{\phi}^{\mathbf{P_C}} = [x]_{\phi}^{\mathbf{P_D}} := c^{\mathbf{P_i}}$. Set $[x]_{\phi}^{\mathbf{P_A}} = [x]_{\phi}^{\mathbf{P_B}} := x - c^{\mathbf{P_i}}$.
    * If $\mathbf{P_i} \in \{\mathbf{P_A}, \mathbf{P_B}\}$, choose $r \xleftarrow{\$} Z_{2^\ell}$, set $[x]_{\phi}^{\mathbf{P_A}} = [x]_{\phi}^{\mathbf{P_B}} := r$. Set $[x]_{\phi}^{\mathbf{P_C}} = [x]_{\phi}^{\mathbf{P_D}} := x - r$.

    Send $(\mathbf{P_j}, \mathsf{sid}, [x]_{\phi}^{\mathbf{P_j}})$ to $\mathbf{P_j}$ for all $\mathbf{P_j} \in \mathcal{P}$. Set $\mathcal{L}_{\mathsf{sid}} = \perp$.
  - Otherwise, set $\mathcal{L}_{\mathsf{sid}} \leftarrow \mathcal{L}_{\mathsf{sid}} \cup \{(\mathsf{chMode}, [x]_{\phi}^{\mathbf{P_i}}, \mathbf{P_i}, \mathsf{sid})\}$.

Fig. 17: Four Party Functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMode}}$.

$\mathcal{F}_{\mathsf{4PC}}^{\mathsf{B2A}}$

Private inputs: Let $\mathcal{P} = \{\mathbf{P_A}, \mathbf{P_B}, \mathbf{P_C}, \mathbf{P_D}\}$. Each party $\mathbf{P_i} \in \mathcal{P}$ holds $\langle x \rangle_{\psi}^{\mathbf{P_i}}$.
Public inputs: Public parameters, ϕ.
Outputs: $[x]_{\phi}^{\mathbf{P_i}}$ for all $\mathbf{P_i} \in \mathcal{P}$.
Initialization: All $\mathcal{L}_{\mathsf{sid}} = \emptyset$.
Compute:

- Upon receiving $(\mathsf{BToA}, \langle x \rangle_{\psi}^{\mathbf{P_i}}, \mathbf{P_i}, \mathsf{sid})$ from the corrupted party $\mathbf{P_i} \in \mathcal{P}$.
- If $\mathcal{L}_{\mathsf{sid}} = \perp$, do nothing.
- Check if every other record $(\mathsf{BToA}, \langle x \rangle_{\psi}^{\mathbf{P_j}}, \mathbf{P_j}, \mathsf{sid})$, $\mathbf{P_j} \in \mathcal{P} \backslash \{\mathbf{P_i}\}$ is in $\mathcal{L}_{\mathsf{sid}}$:
  - If yes, compute $x = \mathsf{rec}([x]_{\psi})$, where $\mathsf{rec}()$ is the reconstruction function.
    If $\phi = \mathsf{AB}$:
    * If $\mathbf{P_i} \in \{\mathbf{P_B}, \mathbf{P_D}\}$, upon receiving $c^{\mathbf{P_i}}$ from $\mathbf{P_i}$, set $[x]_{\phi}^{\mathbf{P_B}} = [x]_{\phi}^{\mathbf{P_D}} := c^{\mathbf{P_i}}$. Set $[x]_{\phi}^{\mathbf{P_A}} = [x]_{\phi}^{\mathbf{P_C}} := x - c^{\mathbf{P_i}}$.
    * If $\mathbf{P_i} \in \{\mathbf{P_A}, \mathbf{P_C}\}$, upon receiving $c^{\mathbf{P_i}}$ from $\mathbf{P_i}$, set $[x]_{\phi}^{\mathbf{P_A}} = [x]_{\phi}^{\mathbf{P_C}} := c^{\mathbf{P_i}}$. Set $[x]_{\phi}^{\mathbf{P_B}} = [x]_{\phi}^{\mathbf{P_D}} := x - c^{\mathbf{P_i}}$.

    If $\phi = \mathsf{AC}$:
    * If $\mathbf{P_i} \in \{\mathbf{P_C}, \mathbf{P_D}\}$, upon receiving $c^{\mathbf{P_i}}$ from $\mathbf{P_i}$, set $[x]_{\phi}^{\mathbf{P_C}} = [x]_{\phi}^{\mathbf{P_D}} := c^{\mathbf{P_i}}$. Set $[x]_{\phi}^{\mathbf{P_A}} = [x]_{\phi}^{\mathbf{P_B}} := x - c^{\mathbf{P_i}}$.
    * If $\mathbf{P_i} \in \{\mathbf{P_A}, \mathbf{P_B}\}$, upon receiving $c^{\mathbf{P_i}}$ from $\mathbf{P_i}$, set $[x]_{\phi}^{\mathbf{P_A}} = [x]_{\phi}^{\mathbf{P_B}} := c^{\mathbf{P_i}}$. Set $[x]_{\phi}^{\mathbf{P_C}} = [x]_{\phi}^{\mathbf{P_D}} := x - c^{\mathbf{P_i}}$.

    Send $(\mathbf{P_j}, \mathsf{sid}, [x]_{\phi}^{\mathbf{P_j}})$ to $\mathbf{P_j}$ for all $\mathbf{P_j} \in \mathcal{P}$. Set $\mathcal{L}_{\mathsf{sid}} = \perp$.
  - Otherwise, set $\mathcal{L}_{\mathsf{sid}} \leftarrow \mathcal{L}_{\mathsf{sid}} \cup \{(\mathsf{BToA}, [x]_{\psi}^{\mathbf{P_i}}, \mathbf{P_i}, \mathsf{sid})\}$.

Fig. 18: Four Party Functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{B2A}}$.