Force: Highly Efficient Four-Party Privacy-Preserving Machine Learning on GPU

Yufan Jiang[1,2], Tianxiang Dai[1], Yong Li[⋆1], Fei Mei[1], Li Duan[1], and Yulian Sun[1]

[1] Huawei European Research Center, Germany
{yufan.jiang,tianxiangdai,yong.li1,fei.mei,li.duan,yulian.sun1}@huawei.com
[2] Karlsruhe Institute of Technology, Germany
yufan.jiang@partner.kit.edu

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 research that Three-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 Four-Party Computation (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 in the *honest-majority* setting. By comparing the results with state-of-the-art research, 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, FantasticFour), CryptGPU and CrypTen.

Keywords: 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 [52]. Technically, it is a good starting point to always keep sensitive

[⋆] Corresponding author

Table 1: Comparison of Force and state-of-the-art works against **semi-honest** adversaries, with max boosting factor.

Setting	Ref.	LAN		WAN
		Training	Inference	Inference
2PC	Cheetah - **CPU**[23]	-	**1234x**	**70x**
	P-SecureML[40, 55]	**14x**	**5.8x**	**13x**
3PC	CryptGPU[50]	**6.5x**	**14x**	**10x**
	P-Falcon[54, 55]	**2.1x**	**3.1x**	**2.4x**
4PC	CrypTen[28]	**38x**	**10x**	**29x**
	P-FantasticFour[12, 55]	**4.7x**	**7.4x**	**10x**
	Force	**1**	**1**	**1**

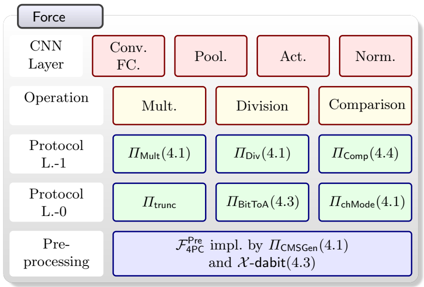

Force
CNN Layer
Conv. FC.
Pool.
Act.
Norm.
Operation
Mult.
Division
Comparison
Protocol L.-1
$\Pi_{\mathsf{Mult}}(4.1)$
$\Pi_{\mathsf{Div}}(4.1)$
$\Pi_{\mathsf{Comp}}(4.4)$
Protocol L.-0
Π_{trunc}
$\Pi_{\mathsf{BitToA}}(4.3)$
$\Pi_{\mathsf{chMode}}(4.1)$
Pre-processing
$\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$ impl. by $\Pi_{\mathsf{CMSGen}}(4.1)$ and $\mathcal{X}$-dabit(4.3)

Fig. 1: Overview of Force protocols

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 [25, 27, 43] 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 [2, 3] in the honest majority setting. Although a giant gap in performance still remains, follow-up works [1, 28, 50] have proposed algorithmic and engineering optimizations to bring MPC closer to real-world and *high-throughput* applications, such as privacy-preserving machine learning (PPML). Thus, a natural question to ask is: *Can 4PC be non-trivially faster than 3PC?* In this paper, we give an affirmative answer. 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 semi-honest secure 2PC/3PC/4PC solutions for privacy-preserving machine learning remarkably by a factor of 2 to 38.

More specifically, we made the following contributions:

- **New 4PC Protocols.** Benefiting from a brand new sharing type called $\mathcal{X}$-share (Section 4.1), our 4PC matrix multiplication protocol achieves the lowest number of local multiplications and number of ring elements sent/received by each party. Besides, we design novel share conversion and comparison protocols with a new type of correlated randomness, the $\mathcal{X}$-dabit (Section 4.3), to achieve the least computational and communication costs. Due to the symmetry property of $\mathcal{X}$-share, we can eliminate the communication in 3PC by turning to the communication-free truncation proposed by SecureML [40] to keep the precision consistent.
- **Extensive Evaluations.** With all the $\mathcal{X}$-share optimized operations as building blocks, shown in Fig. 1, Force greatly improves overall PPML performance. We make fair comparisons between different systems under the same setting. An overview of the evaluation is shown in Table 1. For a better insight, we also include the latest CPU-only framework Cheetah [23].

– **Optimized Graphic Memory Usage.** Unlike [25, 34, 38, 46], which provide inference-only implementations, our aim is to train real-world ML models such as VGG16 [49] with large batch size even for large datasets in MPC over GPU. Given $\mathcal{X}$-share (Section 4.1), Force greatly reduce the graphic memory consumption of each party so that it can perform PPML training of one large dataset, ImageNet [48] on large networks like VGG16 with BatchSize = 16, which was not possible in prior solutions.

2 Related Work

2.1 Privacy Preserving Machine Learning

In 2017, SecureML [40] firstly attempted to execute neural networks (NN) in 2PC, using ABY [13] shares with correlated randomness and mixed protocols with pre-processing. Later attempts like miniONN (2017) [37], secureNN (2019) [53], Falcon (2020) [54], Cheetah (2022) [23] and [22, 25, 34, 38, 39, 46, 56] still follow the mixed protocol approach with various optimization for multiplication and approximation methods for other non-linear operations. Recent 4PC systems [5, 10, 12, 29, 30, 35] also continue with similar approach. All these mainly focus on demonstrating the asymptotic feasibility of PPML and provable security of the system. This might be the primary reason why few of them have taken advantages of GPUs or adapted the solutions for specific ML frameworks.

2.2 PPML on GPU

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

On one hand, crypto researchers turn to GPUs for faster computation. Pu *et al.* [45] in 2011 implemented Yao's Garbled Circuit (GC) [16] on GPU. Later in 2013, Husted *et al.* [24] and Frederiksen and Nielsen [17] worked on more modern GC protocols on GPUs. cuHE [11] brought homomorphic encryption (HE) onto GPU in 2015. These pioneering works uncovered the potential of GPU-friendly MPC, which could be up to 60 times faster than CPU-based solutions [17].

On the other hand, ML researchers pay more attention to privacy. Google in 2016 proposed secure aggregation and Federated Learning (FL) [4] to train shared models over data distributed across users. TensorFlow added support for differential privacy (DP) [14] in 2019 [19]. Although being quite efficient, FL and DP cannot guarantee the same security as MPC does [26, 51].

Finally, the two lines of research meet at CrypTen (2020) [28]. While still having 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 [42] interfaces for ML practitioners, making the PPML framework more approachable for non-cryptographers and extensible for arbitrary number of parties. CryptGPU [50] further extends CrypTen with other GPU-friendly MPC components in a special case: 3PC. In 2022, Watson *et al.* proposed Piranha [55], 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 [49], which was previously impossible on CryptGPU or CrypTen.

For other approaches to implement PPML such as using designated hardware, we refer the reader to the nice surveys [6, 20, 41].

3 Preliminaries

We let a bold $\mathbf{x}$ denote a vector $\mathbf{x} := \{x_0, ..., x_{m-1}\}$ with size m.

3.1 Fixed-point computation

We define a fixed-point value as an ℓ-bit integer using two's complement representation, consisting of both integer part and decimal part with $\ell - p$ bits and p bits respectively. Normally, addition and subtraction will be directly performed over a $\mathbb{Z}_{2^\ell}$ ring, 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-bit decimal precision.

3.2 Correlated Randomness

Correlated randomness are random values with special (algebraic) structural relations that are generated during the pre-processing phase [13, 27, 39, 43] to accelerate the online phase in MPC.

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 a sharing type as $[\cdot]_{\mathsf{RS}}$. 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() with pre-shared keys [50]. 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 Type Conversion :* dabit.** The dabit (doubly authenticated bit) is a type of correlated randomness proposed by Rotaru and Wood [47] to mainly support secure comparison protocol and sharing type conversion. Let $b \xleftarrow{\$} \mathbb{Z}_2$ be the randomness to be shared, $\sum$ the arithmetic sum in the ring, $\oplus$ the binary XOR operation. Formally, dabit is defined as

$$\mathsf{dabit} := ([b], \langle b \rangle)\,, \text{ such that } b = \sum_i [b]_i = \bigoplus_i \langle b \rangle_i,\ [b]_i \xleftarrow{\$} \mathbb{Z}_{2^\ell} \text{ and } \langle b \rangle_i \xleftarrow{\$} \mathbb{Z}_2.$$

***Extended* dabit : edabit.** Recent work [15] of Escudero *et al.* extends dabit to edabit (extended doubly authenticated bit). Similar to dabit, an edabit is a tuple of shares for $b \overset{\$}{\leftarrow} \mathbb{Z}_{2^m}$ defined as

$$\mathsf{edabit} := ([b],\ \langle \mathbf{r} \rangle := (\langle r_0 \rangle, \langle r_1 \rangle, \cdots, \langle r_{m-1} \rangle)),$$

$$\text{such that } b = \sum_i [b]_i,\ [b]_i \overset{\$}{\leftarrow} \mathbb{Z}_{2^\ell} \text{ and } b = \sum_{j=0}^{m-1} r_j 2^j,\ \langle r_j \rangle_i \overset{\$}{\leftarrow} \mathbb{Z}_2.$$

3.3 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 [36] in the honest majority setting often leads to 2PC and 3PC protocols with good efficiency [1, 2, 3, 8, 39, 40, 43, 46, 50, 56], while the ones with *malicious* security [9, 18, 27, 44, 54] are still too heavy for large-scale applications in practice [16]. The **honest majority setting** is also adopted by 4PC frameworks with semi-honest or malicious security [5, 10, 12, 28, 29, 30, 35], 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 see, delay or delete encrypted messages and any non-trivial modification can be detected.

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 $\mathcal{F}$ 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 $\mathcal{Z}$*

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

We follow the universally composable framework (UC) described in detail in [7]. More specifically, we use the hybrid model, where provably UC-secure components are abstracted as ideals in the next proof.

4 4PC Protocols

We construct efficient 4PC protocols as building blocks of Force for PPML. In Section 4.1, we introduce our new sharing type $\mathcal{X}$-share and how parties 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 the best of our knowledge, this becomes the least computation cost compared to other sharing constructions such as replicated or 2-out-of-2 sharing. In Section 4.1 and Section 4.3 we show how to perform conversions between share-modes and sharing types (by using a $\mathcal{X}$-dabit transmitted from dabit [47]).

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

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

$\mathcal{X}$-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 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 $[\cdot]_{\mathsf{AC}}^{\mathbf{P_i}}$ 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 $[\cdot]_{\mathsf{AB}}^{\mathbf{P_i}}$.

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 $x = \sum x_i$.

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 Section 4.1), then locally compute $[x]_{\mathsf{AC}} + [y]_{\mathsf{AC}}$.

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:

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

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 $[z]_{\mathsf{4o4}}^{\mathbf{P_i}} + r^{\mathbf{P_i}}$ to

Protocol $\mathit{\Pi}_{\mathsf{Mult}}$

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. 2: Four party multiplication protocol

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 to the incoming computations.

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 $\mathit{\Pi}_{\mathsf{trunc}}$ introduced by SecureML [40] to avoid the additional communication overhead and round within the truncation protocols $\mathit{\Pi}_{\mathsf{trunc1}}$ and $\mathit{\Pi}_{\mathsf{trunc2}}$ proposed by ABY3 [39]. A detailed description of our multiplication protocol is shown in Fig. 2.

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 $\mathit{\Pi}_{\mathsf{chMode}}$ (Fig. 4) to change the share-mode of either x or y (not both) before multiplication.

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

Suppose parties want to change the share-mode of a shared value 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 $\mathit{\Pi}_{\mathsf{chMode}}$, $\mathbf{P_A}$ and $\mathbf{P_C}$ simply exchange their own 2-out-of-2 sharing masked with r_0 and r_1, obtaining their new shares $x_0+x_1-r_0-r_1$, 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 PRF() with pre-shared keys in the pre-processing stage. We formally define our CMS generation protocol $\mathit{\Pi}_{\mathsf{CMSGen}}$ in Fig. 3, as well as the online protocol $\mathit{\Pi}_{\mathsf{chMode}}$ in Fig. 4.

Division. If parties have to jointly divide a shared value x by a public value γ which is not a power of two, we use the truncation protocol $\mathit{\Pi}_{\mathsf{trunc2}}$ in [39] as a division protocol $\mathit{\Pi}_{\mathsf{Div}}$ to avoid two possible bad events explained in [39].

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

Fig. 3: Four party *changeM share* generation protocol

Π_{Div} consumes a correlated randomness that we call a division share $([r]_\psi, [r']_\phi)$, where $r' = r/\gamma$. The idea behind this protocol is to first reveal $[x]_\psi$ masked with $[r]_\psi$. Parties then compute publicly $(x-r)/\gamma$ and unmask this value by computing $(x-r)/\gamma + [r']_\phi$ locally. Note that we do not require $\phi = \psi$, so the share-mode of the shared division result can be chosen freely.

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 multiplications with ANDs while 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 protocols over boolean inputs [28, 39, 43, 50, 55], while other linear functions (multiplication, convolutions etc.) prefer arithmetic shared values. In the following, we show how conversion between sharing types works, and how parties can determine the share-mode of outputs.

$\mathcal{X}$-dabit. As an important building block, we extend edabit introduced by Escudero *et al.* [15] to $\mathcal{X}$-dabit. Here $b \xleftarrow{\$} \mathbb{Z}_{2^m}$, and ψ and ϕ can be identical.

$$\mathcal{X}\text{-}\mathsf{dabit} := ([b]_\psi, \langle \mathbf{r} \rangle_\phi := (\langle r_0 \rangle_\phi, \cdots, \langle r_{m-1} \rangle_\phi)) \ \text{ s.t. } [b]^{\mathbf{P_i}}_\psi \xleftarrow{\$} \mathbb{Z}_{2^\ell}, \langle r_j \rangle^{\mathbf{P_i}}_\phi \xleftarrow{\$} \mathbb{Z}_2$$

Protocol $\mathit{\Pi}_{\mathsf{chMode}}$

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

Preprocessing: $\mathbf{P_i}$ sends (CMSGen, $\psi\mathsf{to}\phi$ $\mathbf{P_i}, \mathsf{sid}$) to $\mathcal{F}^{\mathsf{Pre}}_{\mathsf{4PC}}$, 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. 4: Four party change share-mode protocol

Protocol $\mathit{\Pi}_{\mathsf{BitToA}}$

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 ($\mathcal{X}$-$\mathsf{dabitGen}$, $\psi, \phi, m, \mathbf{P_i}, \mathsf{sid}$) to $\mathcal{F}^{\mathsf{Pre}}_{\mathsf{4PC}}$ with $m = 1$, receives ($[b]^{\mathbf{P_i}}_\psi$, $\langle b\rangle^{\mathbf{P_i}}_\phi$) 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. 5: Four party bit to arithmetic protocol

To generate $\mathcal{X}$-dabit (in the pre-processing), four parties are assigned into two groups. Then following the protocols proposed by [15] for 2PC setting, each group 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 $\mathit{\Pi}_{\mathsf{chMode}}$ (Fig. 4).

Arithmetic vs. Boolean. We first consider one bit case, where parties have to convert $[x]_\psi$ to $\langle x\rangle_\phi$ with $x \in \mathbb{Z}_2$ (an A2B 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 $[x]^{\mathbf{P_i}}_\psi$ masked with $[b]^{\mathbf{P_i}}_\psi$, then locally "unmask" the revealed value $x - b$ with $\langle b\rangle_\phi$. Converting $\langle x\rangle_\phi$ to $[x]_\psi$ works in the same manner vice versa. A detailed protocol description (B2A for one bit) is placed in Fig. 5. If $x \in \mathbb{Z}_{2^\ell}$, parties can generate an $\mathcal{X}$-dabit with $m = \ell$ to support an A2B protocol, and ℓ pieces $\mathcal{X}$-dabits with $m = 1$ to support a B2A protocol. We refer to [15, 47] for more details.

Table 2: Force compared to the existing works regarding Dot Product (in bits).

Setting	Framework	Preparation	Online		Local	with Trunc	
		Comm	Comm	Rounds	Mult	Comm	Rounds
2PC (S.H.)[3]	P-SecureML [40, 55]	TTP[4]	$4n\ell$	1	3	$4n\ell$	1
3PC (S.H.)	CryptGPU [50]	0	2ℓ	1	3	3ℓ	2
	P-Falcon [54, 55]	0	2ℓ	1	3	4ℓ	1
4PC (S.H.)	CrypTen [28]	TTP	$8n\ell$	1	2	$(8n+4)\ell$	2
	P-FantasticFour [12, 55]	0	4ℓ	1	7	6ℓ	2
	PrivPy [35]	0	4ℓ	1	2	4ℓ	1
4PC (M.)[5]	Trident [10]	3ℓ	4ℓ	1	3	$5\ell(4\ell)$	2(1)
	Swift [29]	3ℓ	3ℓ	2	3	$4\ell(3\ell)$	2(1)
	Tetrad [30]	2ℓ	$4\ell(3\ell)$	2(1)	4	$4\ell(3\ell)$	2(1)
4PC (S.H.)	**Force**	**0**	$\mathbf{2\ell}$	**1**	**1**	$\mathbf{2\ell}$	**1**

4.4 Secure Comparison

We now introduce our secure 4PC comparison protocol Π_{Comp}. We refer the reader to Section 6 for the security of holding both $\langle \mathbf{r} \rangle_\psi$ and $\langle \mathbf{r} \rangle_\phi$ at the same time. For the following computation, we first assume that parties hold ($[b]_\psi$, $\langle \mathbf{r} \rangle_\psi$, $\langle \mathbf{r} \rangle_\phi$), such that $\psi \neq \phi$. Using the same technique mentioned in [55], parties first reveal $[x]_\psi$ by masking it with $[b]_\psi$ (arithmetic part of an $\mathcal{X}$-dabit). After computing the bit decomposition of $x-b$ (we denote those bits as $\mathbf{x}'$), parties can jointly evaluate a parallel prefix adder (PPA) circuit to securely extract the sign bit of $[x]_\psi$. To do so, parties have to prepare a shared propagator $\langle \mathbf{p} \rangle_\psi = \mathbf{x}' \oplus \langle \mathbf{r} \rangle_\psi$ and a shared generator $\langle \mathbf{g} \rangle_\phi = \mathbf{x}' * \langle \mathbf{r} \rangle_\phi$, where $A * B$ denotes a bit-wise AND of A and B, and $\psi \neq \phi$. 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. As a result, the overall AND computations can be executed in a 4PC way.

5 Communication and Computation Analysis

We use DotP to denote the dot product computation (convolution) of two secret vectors, for conciseness. And we let n denote the length of a vector. As already mentioned in Section 4, parties have to rescale (truncate) the shared output of DotP for consistence in precision. A summary of Force and existing works for DotP at each active party (followed by the truncation) is shown in Table 2.

In 2PC, P-SecureML proposed by [40, 55] consumes Beaver Triples to support DotP in the online stage. Instead of implementing a heavy pre-processing computation, P-SecureML simply lets a trusted third party to allocate the shares. Meanwhile, the local truncation technique allows parties to simply truncate the last p bits without any interaction. So the total communication overhead for the online stage is still $4n\ell$ bits.

[3] Semi-Honest

[4] Trusted Third-Party

[5] Malicious

In 3PC, both CryptGPU [50] and P-Falcon [54, 55] use replicated sharing scheme. Parties need to send/receive overall 2ℓ bits after each DotP to reconstruct the replicated share holdings (re-sharing), which yields one communication round. Since a local truncation [40] fails in replicated sharing scheme in 3PC (proven by [39]), parties perform Π_{trunc2} [39] with the help of a pre-computed truncation share $([r], [r'])$, where $r' = r/2^p$. This protocol can be executed combined with re-sharing, which requires parties to exchange 4ℓ bits data in a single communication round. On the other hand, CryptGPU [50] chooses to implement another truncation protocol Π_{Trunc1} [39] to avoid generating truncation share. This results in two rounds and 3ℓ bits communication volume totally.

CrypTen [28] implements 4PC protocols with a 4-out-of-4 sharing scheme. Regardless the triple generation in the pre-processing stage, a party still has to send/receive $8n\ell$ bits and 4ℓ bits within the DotP protocol and the truncation protocol, respectively. Compared to CrypTen, P-FantasticFour [12, 55] uses replicated sharing scheme over four shares, which improves the communication overhead to 6ℓ.

Recently some 4PC protocols such as [10, 29, 30] achieves active security in the honest majority setting (tolerating one malicious corruption). All of those rely on correlated randomnesses generated in the pre-processing stage to accelerate the online computation. While all four parties stay active in the pre-processing stage, some work (such as [29, 30]) choose to activate three parties in the online stage to complete the computation. Parties benefit from having continuous multiplication gates with amortized communication overhead of 3ℓ in one round. We point out that in CNN (e.g. [21, 49]), a convolution layer is followed directly by an activation layer, which requires parties to execute a comparison protocol. As a result, such a construction requires parties to exchange overall 4ℓ elements in two rounds.

Given $\mathcal{X}$-share in Force, we observe a huge computational and communication complexity reduction and a much simplified connection channel establishment. Without relying on a pre-processing stage [6], parties only have to compute one single multiplication locally for DotP. And in fact, parties exchange their local shares with one single partner instead of two, which yields a simpler peer-to-peer connection. Since the local truncation is compatible with $\mathcal{X}$-share, the total communication overhead of Force is only 2ℓ bits in one round.

6 Accelerated Backward in Training

Backward phase is more complicated than the forward phase: for example, it is possible that parties hold a shared x in $[\cdot]_{\mathsf{AC}}$-sharing, which has to be multiplied by two shared values y in $[\cdot]_{\mathsf{AC}}$-sharing and y' in $[\cdot]_{\mathsf{AB}}$-sharing. Yet, it can get accelerated by $\mathcal{X}$-share. First of all, we exclude this situation from the forward phase (except for the comparison protocol), as the computation moves only in one direction without reusing any shared values in multiple computations. The

[6] Recall that generating *Zero Sharing* does not require parties to interact.

easiest way to implement the backward phase is to let parties execute Π_{chMode} if needed. Such a naive solution results in an extra round and communication overhead, but it already has a huge performance improvement compared to other frameworks. A more efficient solution is to let parties hold one shared value in both share-modes, which then enables parties to perform 4PC computations everywhere during the backward phase. Such critical values are normally only weights in each layer, meaning that parties are capable to trade a small portion of memory 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 $x = x'_0 + x'_1$).

7 Evaluation

In this section, we thoroughly evaluate $\mathcal{X}$-share and make in-depth comparisons against other state-of-the-art solutions. We build Force on top of Piranha [55] [7], at commit *bd9c8c4*, in C++. We implement the new 4-party sharing type $\mathcal{X}$-share for all relevant PPML operations. Besides, we add support for batch normalization and complex ResNet like ResNet152, while Piranha only supports layer normalization and basic ResNet18.

7.1 Evaluation Setup

Testbed Environment. We run our evaluations on 4 cloud servers, with 2 CPUs, Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz, and 12×128GB of RAM. Each of our servers is equipped with one GPU, NVIDIA Tesla P100-PCIE with 16GB of video RAM (VRAM). We consider two types of network environments: **LAN** and **WAN**, with 10Gbps bandwidth + $0.2ms$ round-trip latency and 100Mbps bandwidth + $40ms$ round-trip latency, both simulated by the tc tool [8]. Our server is running Ubuntu 18.04.6 LTS with CUDA 10.1.243.

Baseline. We choose as baselines several state-of-the-art systems that have **semi-honest security**, as summarized in Table 1. For 2PC, Cheetah [23] is the most recent PPML work using FHE and correlated oblivious transfer (cOT) on CPUs, which is completely different from ours. We run it on the same server as a baseline of CPU-based PPML. SecureML [40] is the only 2-party system supporting both private inference and training, which is improved by Piranha [55] via porting it to GPU. We refer to the GPU version as P-SecureML. For both 3-party and 4-party, we only consider the **honest-majority** setting. Falcon [54] is the fastest 3-party system on CPU. Piranha [55] ports the **semi-honest** version to GPU with huge boost. We mark it as P-Falcon. CryptGPU [50] is another 3-party system on GPU similar to P-Falcon. We include both of them as baselines. CryptGPU [50] is deployed with the latest Github source code [9], at

[7] https://github.com/ucbrise/piranha/

[8] https://man7.org/linux/man-pages/man8/tc.8.html

[9] https://github.com/jeffreysijuntan/CryptGPU

commit $2ff57b2$. As for 4-party, CrypTen [28] is the only one with **semi-honest security** in an **honest-majority** setting by design. We deploy it using their latest Github source code [10], 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. For fairness, we should not compare with them. Yet, Piranha [55] re-implemented the **semi-honest** version of FantasticFour [12] on GPU. We include this simplified version and refer to it as P-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 [23], which supports maximum 44-bit. All the experiments are performed multiple times, with $\mathsf{BatchSize} = 1$, considering that some systems do not support large batch sizes. Then we calculate the benchmarks by averaging all the results except the first run, to mitigate the influence of system initialization and runtime randomness.

Models and Datasets. For our evaluations, we consider three datasets and three neural networks in different sizes: **Small** ones: CIFAR10 [31] and AlexNet [32]. **Medium** ones: TinyImageNet [33] (Tiny for short) and VGG16 [49]. **Large** ones: ImageNet [48] and ResNet152 [21]. 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 [50] and Falcon [54].

7.2 End-to-End Running Time Evaluation

In Table 3 and 4, we list the running time of an *inference* pass for all datasets and models described in Section 7.1 in *LAN* and *WAN* settings. Our Force completely outperforms all the baseline systems in all evaluations. CPU-based Cheetah is slower than all the other GPU-based systems in *LAN*. In *WAN*, Cheetah (implemented in C++) can perform better than the Python-implemented (CryptGPU and CrypTen) in deep network like ResNet152, while still slower than the C++-implemented (P-SecureML, P-Falcon, P-FantasticFour and Force). When comparing all GPU-based systems, the C++-implemented perform much better than the Python-implemented. This could result from the language performance difference. Among those C++-implemented, Force beats the other three Piranha-based systems, P-SecureML, P-Falcon and P-FantasticFour, in all experiments, with the acceleration brought by our novel sharing type $\mathcal{X}$-share.

Benchmarks of a *training* pass are similar, as shown in Table 6. Cheetah is omitted here as it does not support training. Again, our Force completely dominates in all evaluations.

7.3 Linear vs. Non-Linear Operations

We group common computation tasks into two categories: linear and non-linear. Linear operations include convolution, matrix multiplication and batch normalization. Non-linear operations include ReLU, pooling and SoftMax.

[10] https://github.com/facebookresearch/CrypTen

Table 3: Running time (Second) of an ***inference*** pass in ***LAN***, 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 4: Running time (Second) of an ***inference*** pass in ***WAN***, 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 5: Communication volume (MByte) of an ***inference*** pass, 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 6: Running time (Second) of a ***training*** pass in ***LAN***, BatchSize = 1.

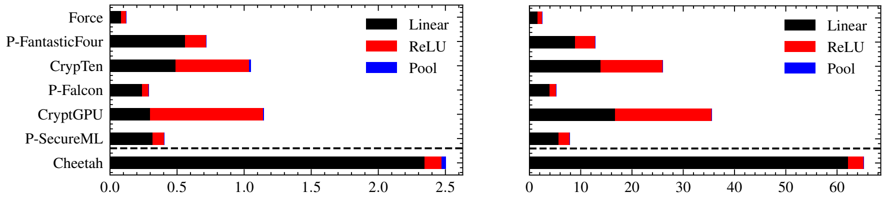
	CIFAR10			Tiny			ImageNet		
	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152
P-SecureML	1.62	4.55	29.20	7.53	5.99	27.81	7.41	28.82	65.51
CryptGPU	2.27	5.49	40.24	3.23	8.06	41.37	9.10	38.86	53.28
P-Falcon	0.75	2.44	12.08	0.96	3.04	13.55	4.13	16.14	35.78
CrypTen	13.48	40.86	27.68	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

Table 7: 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	✓	✓	✓	✓	✓	×

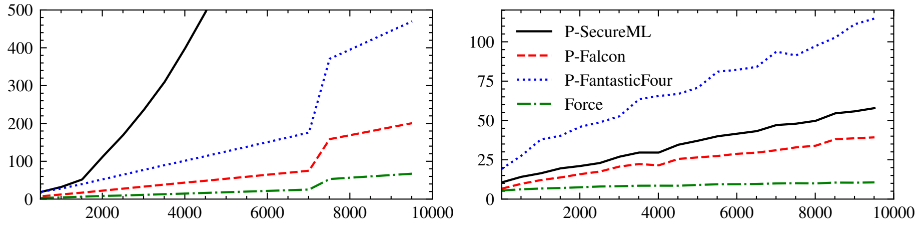
Table 8: Inference accuracy comparison of Force and `PyTorch`.

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



(a) `CIFAR10` + `AlexNet` (b) `CIFAR10` + `ResNet152`

Fig. 6: 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. 7: Micro-benchmark of *matmul* and ReLU in four 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.

We plot the running time of different operations during an *inference* pass in Fig. 6. Due to the huge time difference between Cheetah and all other systems, all the experiments other than the two shown are rarely visible as bar charts. Thus we omit them. We can see that the most time-consuming operation in `Python`-implemented CryptGPU and CrypTen is ReLU, while linear operations cost more in those `C++`-implemented. CryptGPU can be faster than P-SecureML and P-FantasticFour in linear operations, but slower than P-Falcon and Force. The implementation language still makes a difference here.

To further compare the effect of different sharing types, we 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. 7. Force, with the new sharing type $\mathcal{X}$-share, shows outstanding improvement over the other three. Besides, Force scales much better as the problem size increases.

7.4 Communication Cost

One of the main contribution of Cheetah is low communication cost. We make it even better. As shown in Table 5, we have the minimal communication volume

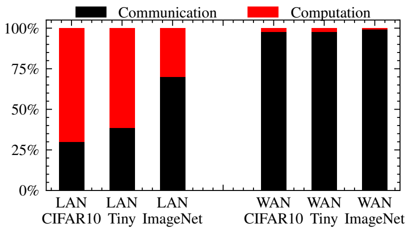


Fig. 8: Time ratio when Force runs ***inference*** on ResNet152.

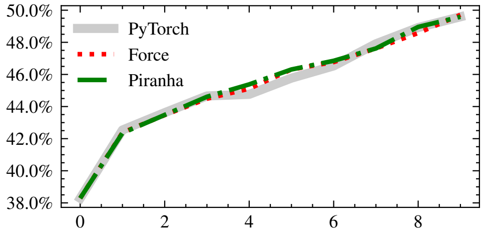


Fig. 9: Validation accuracy of 9 training epochs for AlexNet + CIFAR10.

when performing *inference* with BatchSize = 1 for all the evaluations. 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 Force in Fig. 8. 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.

7.5 Memory Efficiency

Compared with RAM, which could easily reach $1TB$ nowadays, VRAM is an extremely limited resource, which is normally 16GB or 24GB per card. To measure the utilization efficiency of VRAM, we run a simple experiment. We train one large dataset, ImageNet, on one of the large models, VGG16, and try to find out the maximum possible batch size. The result is displayed in Table 7. Force is the only system which supports training ImageNet on VGG16 with BatchSize = 8 and BatchSize = 16. This is achieved by getting rid of Beaver's Triple and reducing the number of local shares to just one. All the other systems can only train with batch size up to 4. CrypTen could not even train with BatchSize = 1.

7.6 Accuracy Comparison

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. The inference accuracy on validation sets is shown in Table 8. Force provides almost the same accuracy as the plaintext PyTorch, only with a tiny relative error of less than 0.1% for all models and datasets. For training, we use AlexNet + CIFAR10 as an example. Starting from a pre-trained model, we train AlexNet on CIFAR10 with Piranha, Force and PyTorch for 9 epochs. The validation accuracy is plotted in Fig. 9. After 9 epochs, the accuracy of PyTorch is 49.59%, of Force is 49.71%, which is even 0.12% higher, indicating extremely low accuracy loss.

8 Conclusion

In this paper, we construct a highly efficient 4PC framework Force for PPML. Our implementation and 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 malicious security and guarantee of delivery, as well as to generalize it for any number of parties.

Bibliography

[1] Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: 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. pp. 805–817 (2016)

[2] Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: European Symposium on Research in Computer Security. pp. 192–206. Springer (2008)

[3] Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. International Journal of Information Security **11**(6), 403–418 (2012)

[4] Bonawitz, K.A., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H.B., Patel, S., Ramage, D., Segal, A., Seth, K.: Practical secure aggregation for federated learning on user-held data. CoRR **abs/1611.04482** (2016)

[5] Byali, M., Chaudhari, H., Patra, A., Suresh, A.: Flash: Fast and robust framework for privacy-preserving machine learning. Proceedings on Privacy Enhancing Technologies **2**, 459–480 (2020)

[6] Cabrero-Holgueras, J., Pastrana, S.: Sok: Privacy-preserving computation techniques for deep learning. Proceedings on Privacy Enhancing Technologies **2021**(4), 139–162 (2021)

[7] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001)

[8] Catrina, O., Hoogh, S.d.: Improved primitives for secure multiparty integer computation. In: International Conference on Security and Cryptography for Networks. pp. 182–199. Springer (2010)

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

[10] Chaudhari, H., Rachuri, R., Suresh, A.: Trident: Efficient 4pc framework for privacy preserving machine learning. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society (2020)

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

[12] Dalskov, A., Escudero, D., Keller, M.: Fantastic four: Honest-majority four-party secure computation with malicious security. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2183–2200 (2021)

[13] Demmler, D., Schneider, T., Zohner, M.: Aby-a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)

[14] Dwork, C., Roth, A., et al.: The algorithmic foundations of differential privacy. Foundations and Trends® in Theoretical Computer Science **9**(3–4), 211–407 (2014)

[15] Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: Improved primitives for mpc over mixed arithmetic-binary circuits. In: Annual International Cryptology conference. pp. 823–852. Springer (2020)

[16] Evans, D., Kolesnikov, V., Rosulek, M., et al.: A pragmatic introduction to secure multi-party computation. Foundations and Trends® in Privacy and Security **2**(2-3), 70–246 (2018)

[17] Frederiksen, T.K., Nielsen, J.B.: Fast and maliciously secure two-party computation using the gpu. In: International Conference on Applied Cryptography and Network Security. pp. 339–356. Springer (2013)

[18] Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: 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. pp. 225–255. Springer (2017)

[19] Google LLC: Tensorflow privacy (2019), `https://github.com/tensorflow/privacy`

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

[21] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)

[22] Hesamifard, E., Takabi, H., Ghasemi, M., Wright, R.N.: Privacy-preserving machine learning as a service. Proc. Priv. Enhancing Technol. **2018**(3), 123–142 (2018)

[23] Huang, Z., Lu, W.j., Hong, C., Ding, J.: Cheetah: Lean and fast secure two-party deep neural network inference. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 809–826 (2022)

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

[25] Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: Gazelle: A low latency framework for secure neural network inference. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1651–1669 (2018)

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

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

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

[29] Koti, N., Pancholi, M., Patra, A., Suresh, A.: SWIFT: Super-fast and robust Privacy-Preserving machine learning. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2651–2668. USENIX Association (Aug 2021)

[30] Koti, N., Patra, A., Rachuri, R., Suresh, A.: 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)

[31] Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images (2009)

[32] Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C., Bottou, L., Weinberger, K. (eds.) Advances in Neural Information Processing Systems. vol. 25. Curran Associates, Inc. (2012)

[33] Le, Y., Yang, X.: Tiny imagenet visual recognition challenge. CS 231N **7**(7), 3 (2015)

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

[35] Li, Y., Xu, W.: Privpy: General and scalable privacy-preserving data mining. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 1299–1307 (2019)

[36] Lindell, Y.: How to simulate it–a tutorial on the simulation proof technique. Tutorials on the Foundations of Cryptography pp. 277–346 (2017)

[37] Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via minionn transformations. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. pp. 619–631 (2017)

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

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

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

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

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

[43] Patra, A., Schneider, T., Suresh, A., Yalame, H.: Aby2. 0: Improved mixed-protocol secure two-party computation. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2165–2182 (2021)

[44] Patra, A., Suresh, A.: BLAZE: blazing fast privacy-preserving machine learning. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society (2020)

[45] Pu, S., Duan, P., Liu, J.: Fastplay-a parallelization model and implementation of SMC on CUDA based GPU cluster architecture. IACR Cryptol. ePrint Arch. p. 97 (2011)

[46] Rathee, D., Rathee, M., Kumar, N., Chandran, N., Gupta, D., Rastogi, A., Sharma, R.: Cryptflow2: Practical 2-party secure inference. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 325–342 (2020)

[47] Rotaru, D., Wood, T.: Marbled circuits: Mixing arithmetic and boolean circuits with active security. In: International Conference on Cryptology in India. pp. 227–249. Springer (2019)

[48] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al.: Imagenet large scale visual recognition challenge. International journal of computer vision **115**(3), 211–252 (2015)

[49] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015)

[50] Tan, S., Knott, B., Tian, Y., Wu, D.J.: Cryptgpu: Fast privacy-preserving machine learning on the gpu. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1021–1038. IEEE (2021)

[51] Truex, S., Baracaldo, N., Anwar, A., Steinke, T., Ludwig, H., Zhang, R., Zhou, Y.: A hybrid approach to privacy-preserving federated learning. In: Proceedings of the 12th ACM workshop on artificial intelligence and security. pp. 1–11 (2019)

[52] United Nations Conference on Trade and Development: Data protection and privacy legislation worldwide (2022), `https://unctad.org/page/data-protection-and-privacy-legislation-worldwide`

[53] Wagh, S., Gupta, D., Chandran, N.: Securenn: 3-party secure computation for neural network training. Proc. Priv. Enhancing Technol. **2019**(3), 26–49 (2019)

[54] Wagh, S., Tople, S., Benhamouda, F., Kushilevitz, E., Mittal, P., Rabin, T.: Falcon: Honest-majority maliciously secure framework for private deep learning. Proceedings on Privacy Enhancing Technologies **1**, 188–208 (2021)

[55] Watson, J.L., Wagh, S., Popa, R.A.: Piranha: A gpu platform for secure computation. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 827–844 (2022)

[56] Zhang, Q., Zhao, Y., Li, L., Zhang, J., Zhang, Q., Zhou, Y., Yin, D., Tan, S., Yin, S.: MORSE-STF: A privacy preserving computation system. CoRR **abs/2109.11726** (2021)