ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation

Arpita Patra
Indian Institute of Science

Thomas Schneider
TU Darmstadt

Ajith Suresh
Indian Institute of Science

Hossein Yalame
TU Darmstadt

Abstract

Secure Multi-party Computation (MPC) allows a set of mutually distrusting parties to jointly evaluate a function on their private inputs while maintaining input privacy. In this work, we improve semi-honest secure two-party computation (2PC) over rings, with a focus on the efficiency of the online phase.

We propose an efficient mixed-protocol framework, outperforming the state-of-the-art 2PC framework of ABY. Moreover, we extend our techniques to multi-input multiplication gates without inflating the online communication, i.e., it remains independent of the fan-in. Along the way, we construct efficient protocols for several primitives such as scalar product, matrix multiplication, comparison, maxpool, and equality testing. The online communication of our scalar product is two ring elements *irrespective* of the vector dimension, which is a feature achieved for the first time in the 2PC literature.

The practicality of our new set of protocols is showcased with four applications: i) AES S-box, ii) Circuit-based Private Set Intersection, iii) Biometric Matching, and iv) Privacy-preserving Machine Learning (PPML). Most notably, for PPML, we implement and benchmark training and inference of Logistic Regression and Neural Networks over LAN and WAN networks. For training, we improve online runtime (both for LAN and WAN) over SecureML (Mohassel et al., IEEE S&P'17) in the range 1.5×–6.1×, while for inference, the improvements are in the range of 2.5×–754.3×.

1 Introduction

Secure Multi-Party Computation (MPC) [13, 45, 98] allows n mutually distrusting parties to jointly compute a function on their private inputs. The computation guarantees i) privacy– no set of t corrupt parties can learn more information than the output, and ii) correctness– corrupt parties cannot force others to accept a wrong output. Due to its immense potential, MPC can be used for solving real-life applications such as privacy-preserving auctions [77] and remote diagnostics [23], secure genome analysis [14, 96], and recently in the domain of privacy-preserving machine learning (PPML) [16, 27, 30, 31, 52, 57, 67, 75, 84, 91, 101].

MPC protocols can be broadly classified into low-latency [28, 46, 74, 81, 97] and high-throughput [4, 27, 30, 31, 67, 84] categories. The low-latency protocols are built using Yao's garbled circuits (GC) [10, 66, 97–99] and result in constant-round solutions. Secret-sharing (SS) based solutions have been used for high-throughput protocols, but require a number of communication rounds linear in the multiplicative depth of the circuit. However, less communication than GC-based protocols facilitates several instances of SS-based protocols to be executed in parallel, leading to high throughput. The characteristics of the categories mentioned above put forth the need for a mixed-protocol framework [31, 39, 73, 75, 92], where the protocol is split into blocks and each block is executed in one of the following three worlds: i) Arithmetic, ii) Boolean, and iii) Yao. While the arithmetic world performs operations on ℓ-bit rings (or fields), both boolean and Yao world perform operations on bits. Also, arithmetic and boolean worlds operate using an SS-based approach while the Yao world uses a GC-based approach.

To achieve practical runtimes, several works [12, 26, 27, 30, 31, 38, 61, 67, 91] considered the paradigm of having an *input-independent* setup phase where the parties generate a lot of correlated randomness (e.g., Beaver multiplication triples [8]) which are then used in the *input-dependent* online phase to enable a very fast computation on the parties' inputs. Moreover, the benchmarking results of [94] and the works of [17, 34, 35, 37, 39] have showcased the efficiency improvements of protocols compared to rings over their field counterparts. The 32/64-bit computations done in standard CPUs, emulating ring operations, allow for very simple and efficient implementations.

In this work, we focus on the specific problem of secure two-party computation (2PC) [38, 39] with mixed protocols over rings. Our aim is to minimize the online communication and rounds keeping high throughput as our end-goal.

1.1 Our Contributions

We propose an efficient mixed-protocol framework for secure 2PC over an ℓ-bit ring. Our protocols are secure against a *semi-honest* adversary and use an *input-independent* setup. We build several building blocks with the focus on online efficiency. Our contributions can be summed up as follows:

2PC (§3). We propose an efficient 2PC protocol over ℓ-bit rings, requiring a communication of just 2 ring elements per multiplication in the online phase. Our construction relies on Beaver's circuit randomization technique [8] (§3.1.1), but uses a different perspective of the technique. Moreover, our protocol helps in realising efficient primitives as will be shown in §5. We believe that our new perspective can bring several further optimizations where Beaver's randomization technique is currently being used.

Protocol	Ref.	Setup	Online	
		Comm [bits]	Comm [bits]	Rounds
MULT $z = ab$	[39]	$2\ell(\kappa+\ell)$	4ℓ	**1**
	[12]	$2\ell(\kappa+\ell)$	$\mathbf{2\ell}$	**1**
	[78]	$2\ell(\kappa+\ell)$	4ℓ	**1**
	ABY2.0	$2\ell(\kappa+\ell)$	$\mathbf{2\ell}$	**1**
MULT3 $z = abc$	[39]	$4\ell(\kappa+\ell)$	8ℓ	2
	[12]	$4\ell(\kappa+\ell)$	4ℓ	2
	[78]	$8\ell(\kappa+\ell)$	6ℓ	**1**
	ABY2.0	$8\ell(\kappa+\ell)$	$\mathbf{2\ell}$	**1**
MULT4 $z = abcd$	[39]	$6\ell(\kappa+\ell)$	12ℓ	2
	[12]	$6\ell(\kappa+\ell)$	6ℓ	2
	[78]	$22\ell(\kappa+\ell)$	8ℓ	**1**
	ABY2.0	$22\ell(\kappa+\ell)$	$\mathbf{2\ell}$	**1**

Table 1: Comparison of ABY2.0 and existing works for 2PC protocols. Best values for the online phase are marked in bold.

Tab. 1 shows our improvement over previous works. For 2-input multiplication, we achieve the same complexity as [12], but using a completely different approach. Moreover, for an N-input multiplication gate, our solution has a *constant* cost of 2 ring elements and one round of interaction. This is a massive improvement over [78], where they require communication of $2N$ ring elements. Round complexity wise, the naive method of multiplying N elements by taking two at a time requires $\log_2(N)$ online rounds and overall communication of $4(N-1)$ ring elements for [39] and $2(N-1)$ for [12].

Mixed Protocol Conversions (§4). The mixed world conversions, that enable easy transition between Arithmetic (**A**), Boolean (**B**) and Yao (**Y**) sharing, are now celebrated in the literature [3, 26, 57, 75, 91] due to their potential in building practically-efficient protocols. We propose a new set of conversions that outperform the state-of-the-art conversions of ABY [39] in the online phase. Our solution reduces the number of online rounds of ABY from 2 to 1 for most of the conversions. We achieve this because, in contrast to ABY, we forgo OTs in the online phase of our conversions.

Tab. 2 provides the concrete costs for the mixed protocol conversions. The conversion from sharing type S to sharing type D is denoted as $S2D$, where $S, D \in \{\mathbf{A}, \mathbf{B}, \mathbf{Y}\}$. For the setup phase, we use correlated OTs (cOT) [5] which incur a communication of $\ell+\kappa$ bits per cOT on ℓ-bit strings, where κ is the computational security parameter. It is evident from Tab. 2 that for all except the Y2B conversion, our conversions outperform ABYs' in the online phase.

Conv.	Ref.	Setup	Online	
		Comm [bits]	Comm [bits]	Rounds
Y2B	ABY [39]	0	**0**	**0**
	ABY2.0	ℓ	ℓ	1
B2Y	ABY [39]	$2\ell\kappa$	$\ell\kappa+\ell$	2
	ABY2.0	$2\ell\kappa$	$\boldsymbol{\ell\kappa}$	**1**
A2Y	ABY [39]	$4\ell\kappa$	$2\ell\kappa+\ell$	2
	ABY2.0	$4\ell\kappa$	$\boldsymbol{\ell\kappa}$	**1**
Y2A	ABY [39]	$2\ell\kappa$	$(\ell^2+3\ell)/2$	2
	ABY2.0	$3\ell\kappa+2\ell$	$\boldsymbol{\ell}$	**1**
A2B	ABY [39]	$4\ell\kappa$	$2\ell\kappa+\ell$	**2**
	ABY2.0	$4\ell\kappa+\ell$	$\boldsymbol{\ell\kappa+\ell}$	**2**
B2A	ABY [39]	$\ell\kappa$	$(\ell^2+\ell)/2$	2
	ABY2.0	$\ell\kappa+\ell^2$	$\mathbf{2\ell}$	**1**

Table 2: Comparison of ABY2.0 and ABY for the conversions. The values are reported for ℓ-bit values. Best values for the online phase are marked in bold.

Building Blocks (§5). We propose efficient constructions for widely-used building blocks that include Scalar Product, Depth-Optimized Circuits, Matrix Multiplication, Comparison, Non-linear Activation functions, and Maxpool. The highlights include:

– *Scalar Product (§5.1):* Our new protocol incurs an online communication that is *independent* of the vector dimension n. This feature is achieved for the first time in the 2PC literature. Concretely, we require communication of just 2 ring elements as opposed to $4n$ elements of [39]. Since scalar product forms an essential building block for most of the widely used ML algorithms [27, 30, 31, 56, 73, 75, 91] such as Linear Regression, Logistic Regression, and Clustering, our solution substantially improves the performance of their secure 2PC implementations by several orders of magnitude.

– *Matrix Multiplication (§5.2):* Matrix multiplication is the fundamental building block in most ML algorithms. For instance, the linear layer in a Neural Network (NN) as well as the convolution operation in a Convolutional Neural Network [95] can be viewed as an instance of matrix multiplication. We extend the 2PC multiplication protocol to support vector operations and provide an efficient matrix multiplication protocol.

– *Depth Optimized Circuits (§5.3):* The Parallel Prefix Adder (PPA) [7, 47] used in the recent PPML literature [73] incurs a multiplicative depth of $\log_2(\ell)$ since it uses two-input AND gates only. We propose round efficient PPA constructions using a combination of two, three, and four input AND gates.

For a 64-bit ring, our solution has 2× fewer rounds and also less online communication compared to the PPA used in [73].

– *Comparison (§5.4):* Our new protocol for checking less than relation improves the online communication of the comparison protocol of [78] by 6× and reduces the number of online rounds from 4 to 3.

– *Maximum of three elements (§5.7):* Our new protocol improves the online communication of [78] by 14× while reducing the online rounds from 5 to 4.

– *Equality Test (§5.10):* Our new protocol for checking the equality of two ℓ-bit values, improves the online rounds of [87] from $\log_2(\ell)$ to $\log_4(\ell)$.

Applications (§6). The practicality of our constructions are showcased in these four popular applications:

– *AES S-box (§6.2):* Using our protocol for 3-input multiplication, we obtain an S-box with an AND-depth of 3 instead of 4 before. This improves the online round complexity of AES by factor 1.33×.

– *Circuit-based PSI (§6.3):* Using our efficient equality testing protocol, we improve the online communication of the state-of-the-art circuit-based PSI [87] by 2.35× and the online round complexity by 1.3×.

– *Biometric Matching (§6.4):* We propose a round-optimized as well as a communication-optimized solution for computing the minimum Euclidean distance, which forms the core for biometric matching. For the round-optimized variant, we improve over ABY [39] by 2.2× in communication and 1.6× in rounds in the online phase. Similarly, for the communication-optimized variant, we improve over [78] by 20.8× in communication and 1.3× in rounds.

– *Privacy-Preserving Machine Learning (§6.5):* Here we implement the training and inference of Logistic Regression and Neural Networks in a LAN and a WAN setting and benchmarked over datasets with various feature sizes.

Algorithm	Ref.	LAN		WAN	
		TP (x10^4)	Improvem.	TP (x10^4)	Improvem.
Logistic Regression	[75]	1,344.4	**31.5×**	4.0	**9.9×**
	ABY2.0	**42,372.4**		**39.9**	
Neural Networks	[75]	43.0	**716.0**×	0.1	**710.7**×
	ABY2.0	**30,797.0**		**92.39**	

Table 3: Comparison of the online throughput (TP) of ABY2.0 and SecureML [75] for inference on the MNIST [70] dataset.

For training, we obtain online runtime improvements over SecureML [75] in the range 2.7×–6.1× for LAN and 1.5×–2.8× for WAN. For inference, we used *throughput* as one metric to capture the effect of runtime and communication utilization in a single shot. Our improvement for inference ranges from 7.9×–754.3× for LAN, while it ranges from 2.5×–753.2× for WAN. Tab. 3 provides the concrete details for inference over the MNIST [70] dataset.

1.2 Related Work

Here, we provide a concise summary of related work. More details on the preliminaries are given in §A.

Secret Sharing (SS). The works of [38,61] proposed efficient SS-based solutions for the dishonest majority setting over fields, which was then extended to the ring setting in [33]. The solution involves the generation of Beaver multiplication triples [8] in the setup phase and evaluation of the circuit (multiplication gates) in the online phase using the generated triples. For the 2PC case, the aforementioned approach requires two public reconstructions among the parties per multiplication gate in the online phase. In contrast, we require only one public reconstruction among the parties. Later, works like [59, 60, 79] focused on improving the setup cost using techniques like Oblivious Transfer (OT) and Homomorphic Encryption (HE). [12] improved the number of public reconstructions required in the online phase from two to one using a function-dependent preprocessing, but requires additional communication of four ring elements in the preprocessing phase.

Multi-Input Multiplication. In the boolean setting, [40] extended two-input AND gates to the general N-input case using lookup tables. Recently, [78] extended the multiplication from two-input to arbitrary input using Beaver triple extension with a focus on minimizing the online rounds. However, the online communication of [78] scale with the fan-in of the multiplication gates as opposed to ours, where we achieve an online communication of 2 ring elements.

Mixed-Protocol Conversions. Mixed 2PC protocols that combine GC-based and SS-based approaches benefit from their respective advantages and were used in many privacy-preserving applications such as face recognition [49], fingerprint recognition [24], biometric matching [39], and machine learning [57, 73, 75, 91]. The first mixed-protocol framework for MPC was TASTY [49, 65], which combined garbled circuits with homomorphic encryption. ABY [39] then proposed an efficient framework in the semi-honest model combining state-of-the-art 2PC approaches based on Arithmetic sharing, Boolean sharing, and GCs. The work of [92] shows conversions between MPC based on arithmetic secret sharing and garbled circuits with malicious security. Later, the ABY framework was extended to the three and four party honest-majority setting by [31, 73]. HyCC [26] provides a compiler to automatically partition a function (specified in ANSI C) into sub-functions such that each sub-function is evaluated with either Arithmetic sharing, Boolean sharing or Garbled Circuits (GC).

2 Preliminaries

Here, we describe our security model and the parameters and notations used. More details along with a brief overview of the state-of-the-art 2PC protocols are given in §A.

Semi-honest Security Model. In this work, we consider a semi-honest (aka passive) adversary [32, 53, 100], who is "honest-but-curious". The adversary is guaranteed to follow the protocol steps but will try to learn additional information from the messages that he has seen during the protocol execution. Though not the strongest model, this model forms the first step towards achieving protocols with stronger security guarantees [6, 29, 68, 71]. Also, the setting facilitates practically-efficient protocols with higher performance especially for PPML applications [30, 75, 91]. In practical scenarios where the computation is outsourced to a set of servers, the reputation of the servers forces them to behave semi-honestly. Moreover, in many application scenarios, semi-honest behaviour can be enforced by attestation using tools like Intel SGX or ARM TrustZone. We refer the reader to [44] for details on the model.

Parameters and Notation. In our framework, we have two parties $\mathcal{P} = \{P_0, P_1\}$ who are connected by a bidirectional synchronous channel (eg. instantiated via TLS over TCP/IP). Our protocols are designed to work over an ℓ-bit ring denoted by $\mathbb{Z}_{2^\ell}$. κ denotes the computational security parameter. In our implementation, we use $\ell = 64$ and $\kappa = 128$.

For two vectors $\vec{a}, \vec{b}$ of length n, the scalar dot product is denoted by $\vec{a} \odot \vec{b} = \sum_{j=1}^{\mathsf{n}} \mathsf{a}_j \mathsf{b}_j$. Here a_j and b_j denote the j^{th} elements of vectors $\vec{a}$ and $\vec{b}$ respectively. For a bit $\mathsf{u} \in \{0,1\}$, $\overline{\mathsf{u}}$ denotes the complement value $1 \oplus \mathsf{u}$. For two matrices $\mathbf{A}, \mathbf{B}$, matrix multiplication is denoted by $\mathbf{A} \circ \mathbf{B}$. Table 4 depicts notation that we use throughout the paper.

P_0, P_1	Parties performing secure computation
$\mathbb{Z}_{2^\ell}$	Ring of size ℓ bits; $\ell = 64$ in this work
κ	Symmetric security parameter; $\kappa = 128$ in this work
a_j	j-th element of vector $\vec{a}$
$\vec{a} \odot \vec{b}$	Scalar dot product between two vectors $\vec{a}$ and $\vec{b}$
$\mathbf{A} \circ \mathbf{B}$	Multiplication of two matrices $\mathbf{A}$ and $\mathbf{B}$
$[\mathsf{v}]_i$	$[\cdot]$-sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ held by P_i s.t. $\mathsf{v} = [\mathsf{v}]_0 + [\mathsf{v}]_1$
$\langle \mathsf{v} \rangle_i = ([\delta_\mathsf{v}]_i, \Delta_\mathsf{v})$	$\langle \cdot \rangle$-sharing of $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ held by P_i s.t. $\mathsf{v} = \Delta_\mathsf{v} - [\delta_\mathsf{v}]_1 - [\delta_\mathsf{v}]_0$
$t \in \{A, B, Y\}$	Type of sharing: **A**rithmetic, **B**oolean, or **Y**ao
$x^s = s2t(x^t)$	Sharing conversion from source s to target t
OT	Oblivious Transfer
HE	Homomorphic Encryption
cOT_ℓ^n	n instances of Correlated OT on ℓ-bit strings
MSB/LSB	Most / Least Significant Bit
FPA	Fixed-point Arithmetic
SED	Squared Euclidean Distance

Table 4: Notations used throughout this paper.

Our protocols are cast into an *input-independent* setup phase and an *input-dependent* online phase. To enable parties to non-interactively sample a random value, parties perform a one-time key-setup that establishes random keys among them for a pseudo-random function (PRF) which can be instantiated, for instance, using AES in counter mode. Towards this, each party P_i for $i \in \{0,1\}$ samples a random key $K_i \in_R \{0,1\}^\kappa$ and sends it to the other party. The shared key is now defined as $K = K_0 + K_1$.

For applications such as machine learning where the inputs are decimal numbers, we use the Fixed-Point Arithmetic (FPA) representation [27, 30, 31, 73, 75] to embed the value in the underlying ring. Decimal value is treated as an ℓ-bit integer in signed 2's complement representation. The most significant bit (MSB) represents the sign while the least significant x bits represent the fractional part. For our implementation, we use $\ell = 64$ and $x = 13$.

3 2PC in Arithmetic, Boolean & Yao World

The contribution of this section is our new 2PC over ring $\mathbb{Z}_{2^\ell}$. This construction gives us a new 2PC in the arithmetic world and in the Boolean world. The latter is easily derived by having $\ell = 1$. The 2PC in Yao's world is borrowed from ABY [39]. Below, we start with our new 2PC over $\mathbb{Z}_{2^\ell}$. We describe the secret-sharing semantics, the sharing and reconstruction protocols, and the multiplication protocols (both for setup and online phase) with various fan-ins. Our final 2PC for any functionality represented over an arithmetic circuit over $\mathbb{Z}_{2^\ell}$ can be obtained by running the following steps in sequence: (a) sharing all the inputs via the sharing protocols, (b) gate by gate evaluation (using linearity of our secret sharing and the multiplication protocols) and (c) output reconstruction via the reconstruction protocol.

3.1 2PC in Arithmetic World

We provide the details for our 2PC scheme here. Before going into the details, we present a high-level overview of our scheme and a side-by-side comparison with the well-known Beaver's circuit randomization technique [8]. Our protocol, inspired by the 3PC protocol of ASTRA [30], achieves a communication similar to [12]. The highlight of our protocol is its effectiveness towards efficient realisations for multiple input multiplication gates and dot product operations as will be explained in §3.1.4 and §5.1 later.

3.1.1 High-level Overview of Our 2PC over Ring

Consider two parties P_0, P_1 with values a, b additively shared among them who want to compute a multiplication gate with output $\mathsf{c} = \mathsf{a} \cdot \mathsf{b}$.

Beaver's Technique [8] on Gate Inputs (cf. left of Fig. 1). In 2PC, there has been a lot of works [38, 39, 57, 61, 91] that use Beaver's [8] circuit randomization technique to compute the product $\mathsf{a} \cdot \mathsf{b}$. In this technique (cf. left side of Fig. 1), the inputs of the multiplication gate are randomized first and the corresponding correlated randomness is generated independently (preferably in a setup phase). In detail, parties interactively generate an additive sharing of the multiplication triple $(\delta_\mathsf{a}, \delta_\mathsf{b}, \delta_\mathsf{ab})$ with $\delta_\mathsf{ab} = \delta_\mathsf{a}\delta_\mathsf{b}$ during the setup phase before the actual inputs are known. Now, we can write

$$\begin{aligned} \mathsf{a} \cdot \mathsf{b} &= ((\mathsf{a} + \delta_\mathsf{a}) - \delta_\mathsf{a})((\mathsf{b} + \delta_\mathsf{b}) - \delta_\mathsf{b}) \\ &= (\mathsf{a} + \delta_\mathsf{a})(\mathsf{b} + \delta_\mathsf{b}) - (\mathsf{a} + \delta_\mathsf{a})\delta_\mathsf{b} - (\mathsf{b} + \delta_\mathsf{b})\delta_\mathsf{a} + \delta_\mathsf{ab}. \end{aligned}$$

Let $\Delta_a = (a + \delta_a)$ and $\Delta_b = (b + \delta_b)$ be the randomized versions of the input values of a multiplication gate. Then, during the online phase, parties locally compute an additive sharing of Δ_a using additive shares of a and δ_a. Similarly, an additive sharing of Δ_b is computed. This is followed by the parties mutually exchanging the shares of Δ_a and Δ_b to enable public reconstruction of Δ_a and Δ_b. Then using the above equation, parties can locally compute a sharing of $a \cdot b$. Note that this method requires communicating 4 elements per multiplication (2 elements per reconstruction). We observe that the communication is required for enabling parties to obtain the value of Δ_a and Δ_b in clear.

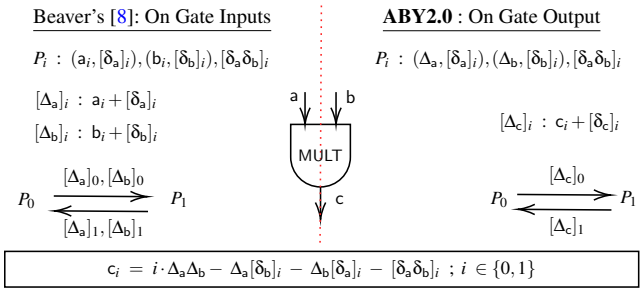

Figure 1: High level overview of Beaver's [8] and **ABY2.0**

Our Technique on Gate Outputs (cf. right of Fig. 1). With this insight, we modify the sharing semantics so that the parties are ensured to have the Δ value as a part of their share, corresponding to every wire value (including the inputs of a multiplication gate). As a result, the reconstructions of Δ_a and Δ_b are no longer required. This may give the wrong impression that no communication is required for evaluating a multiplication gate. It is true that now the parties can locally evaluate the additive sharing of $c = a \cdot b$. But in order to proceed further, a sharing for c according to the new sharing semantics needs to be generated. This requires both parties to obtain Δ_c in clear. Hence, the parties locally compute an additive sharing of Δ_c using the shares of c computed earlier and mutually exchange their shares to reconstruct Δ_c.

Our technique, in summary, shifts the need of reconstruction (which alone causes communication for a multiplication gate) from per input wire to the *output* wire alone for a multiplication gate. For a traditional 2-input multiplication gate, we reduce the number of reconstructions (each involves sending 2 elements) from 2 to 1. As a result, we improve communication by a factor of $2\times$. The impact is much higher for an N-input multiplication gate (cf. §3.1.4) and a scalar product of two N-dimensional vectors (cf. §5.1). For scalar product, Beaver's circuit re-randomization required $2N$ reconstructions, whereas our techniques need a *single* one, offering a gain of $2N\times$. Our constructions can be generalized to the n-party scenario (which is out of scope for this work) and bring a significant pay-off, as the cost per reconstruction depends linearly on the number of parties.

3.1.2 Sharing Semantics

$[\cdot]$-sharing. A value $v \in \mathbb{Z}_{2^\ell}$ is said to be $[\cdot]$-shared among $\mathcal{P}$, if party P_i for $i \in \{0,1\}$ holds $[v]_i$ such that $v = [v]_0 + [v]_1$.

$\langle\cdot\rangle$-sharing. A value $v \in \mathbb{Z}_{2^\ell}$ is said to be $\langle\cdot\rangle$-shared among $\mathcal{P}$, if there exist values $\delta_v, \Delta_v \in \mathbb{Z}_{2^\ell}$ such that i) δ_v is $[\cdot]$-shared among P_0, P_1, ii) $\Delta_v = v + \delta_v$, and iii) Δ_v is known to both P_0, P_1 in clear. We denote the shares of individual parties as $\langle v\rangle_i = ([\delta_v]_i, \Delta_v)$ for $i \in \{0,1\}$.

We use $\delta_{v_1 \ldots v_n}$ to represent the product $\delta_{v_1}\delta_{v_2}\cdots\delta_{v_n}$. Similarly, $\Delta_{v_1 \ldots v_n}$ represents $\Delta_{v_1}\Delta_{v_2}\cdots\Delta_{v_n}$.

3.1.3 Protocols

Sharing Protocol. Protocol SHARE enables party P_i for $i \in \{0,1\}$ to generate a $\langle\cdot\rangle$-sharing of its input value v. During the setup, P_i samples random $[\delta_v]_i$ while the parties together sample $[\delta_v]_{1-i}$ so that P_i will get to know $\delta_v = [\delta_v]_0 + [\delta_v]_1$ in clear. During the online phase, P_i computes $\Delta_v = v + \delta_v$ and sends it to P_{1-i}.

Reconstruction Protocol. To reconstruct value v given $\langle v\rangle$, protocol REC proceeds as follows: parties mutually exchange their missing $[\cdot]$-share of δ_v and locally compute $v = \Delta_v - [\delta_v]_0 - [\delta_v]_1$.

Linear Operations. Our sharing scheme is linear in the sense that given $\langle a\rangle, \langle b\rangle$ and public constants c_1, c_2, parties can locally compute $\langle y\rangle = c_1 \cdot \langle a\rangle + c_2 \cdot \langle b\rangle$. For this, P_i for $i \in \{0,1\}$ locally sets $\Delta_y = c_1 \cdot \Delta_a + c_2 \cdot \Delta_b$ and $[\delta_y]_i = c_1 \cdot [\delta_a]_i + c_2 \cdot [\delta_b]_i$.

Protocol $\mathsf{MULT}(\langle a\rangle, \langle b\rangle)$

Setup:

- P_i for $i \in \{0,1\}$ samples random $[\delta_y]_i \in_R \mathbb{Z}_{2^\ell}$.
- Parties execute $\mathsf{setupMULT}([\delta_a], [\delta_b])$ to generate $[\delta_{ab}]$.

Online:

- P_i for $i \in \{0,1\}$ locally computes and sends to P_{1-i} $[\Delta_y]_i = i \cdot \Delta_{ab} - \Delta_a[\delta_b]_i - \Delta_b[\delta_a]_i + [\delta_{ab}]_i + [\delta_y]_i$.
- P_i for $i \in \{0,1\}$ locally sets $\Delta_y = [\Delta_y]_0 + [\Delta_y]_1$.

Figure 2: Multiplication Protocol

Multiplication Protocol. Given the $\langle\cdot\rangle$-sharing of a, b, the goal of protocol MULT (cf. Fig. 2) is to generate $\langle y\rangle$ where $y = ab$. For correctness to hold, we will need

$$\begin{aligned}\Delta_y = y + \delta_y = ab + \delta_y &= (\Delta_a - \delta_a)(\Delta_b - \delta_b) + \delta_y \\ &= \Delta_a\Delta_b - \Delta_a\delta_b - \Delta_b\delta_a + \delta_a\delta_b + \delta_y.\end{aligned}$$

Since the δ-values are not available in clear to any of P_0, P_1, they cannot compute the value Δ_y on their own. But if we enable the parties obtain a $[\cdot]$-sharing of $\delta_{ab} = \delta_a\delta_b$, then each of them can compute a $[\cdot]$-sharing of Δ_y which they can mutually exchange to obtain Δ_y in clear. So the problem of multiplication reduces to generating $[\delta_{ab}]$ given $[\delta_a]$ and

$[\delta_b]$. We use protocol setupMULT to accomplish this task, the details of which is provided later in this subsection. We note that Turbospeedz [12] achieves same online cost as that of ours, but with a more expensive preprocessing. We provide more details in §A.3.

To summarize, during the setup phase, parties first locally sample the $[\cdot]$-shares for δ_y. In parallel, parties execute the setupMULT protocol on $[\delta_a]$ and $[\delta_b]$ to obtain $[\delta_{ab}]$. During the online phase, the parties locally compute $[\Delta_y]$ and subsequently reconstruct Δ_y.

We now provide the details for instantiating setupMULT using two of the well-known primitives: i) Oblivious Transfer (OT) as used in [39, 59] and ii) Homomorphic Encryption (HE) as used in [38,49,90]. These two approaches have been rallied against each other in terms of practical efficiency in the past and fair competition is still going on. In our work, we make only black-box access to these primitives, and hence an improvement in any of them will have a direct impact on the efficiency of the setup phase of our protocols.

Note that $\delta_{ab} = ([\delta_a]_0 + [\delta_a]_1)([\delta_b]_0 + [\delta_b]_1) = [\delta_a]_0[\delta_b]_0 + [\delta_a]_0[\delta_b]_1 + [\delta_a]_1[\delta_b]_0 + [\delta_a]_1[\delta_b]_1$. Here P_i for $i \in \{0,1\}$ can locally compute $[\delta_a]_i[\delta_b]_i$ and hence the problem reduces to computing $[\delta_a]_0[\delta_b]_1$ and $[\delta_a]_1[\delta_b]_0$.

OT based setupMULT. In our OT-based approach, we use Correlated OTs (cOT) [5] where the sender inputs a correlation function $f(\cdot)$ to cOT and obtains (m_0, m_1), where m_0 is a random element and $m_1 = f(m_0)$. We use cOT_ℓ^n to represent n parallel instances of 1-out-of-2 Correlated OTs on ℓ bit input strings.

To compute $[([\delta_a]_0[\delta_b]_1)]$, the parties execute cOT_ℓ^ℓ with P_0 being the sender and P_1 being the receiver. For the j-th instance of cOT where $j \in \{0,\ldots,\ell-1\}$, P_0 inputs the correlation $f_j(x) = x + 2^j[\delta_a]_0$ and obtains $(m_{j,0} = r_j, m_{j,1} = r_j + 2^j[\delta_a]_0)$. P_1 inputs choice bit b_j as the j-th bit of $[\delta_b]_1$ and obtains m_{j,b_j} as output. Now the $[\cdot]$-shares are defined as $[([\delta_a]_0[\delta_b]_1)]_0 = \sum_{j=0}^{\ell-1}(-r_j)$ and $[([\delta_a]_0[\delta_b]_1)]_1 = \sum_{j=0}^{\ell-1} m_{j,b_j}$. Computation of $[([\delta_a]_1[\delta_b]_0)]$ proceeds similarly with the role of the parties reversed.

HE-based setupMULT. In a HE based solution, P_0, using his public key pk_0, encrypts its messages $[\delta_a]_0, [\delta_b]_0$ in independent ciphertexts and sends the ciphertexts to P_1. In parallel, P_1 computes the ciphertexts corresponding to $[\delta_a]_1, [\delta_b]_1$ and a random element $r \in_R \mathbb{Z}_{2^\ell}$ using pk_0. Upon receiving the ciphertexts from P_0, P_1 computes the ciphertext corresponding to $\mathsf{v} = [\delta_a]_0[\delta_b]_1 + [\delta_a]_1[\delta_b]_0 - r$ using the homomorphic property of the underlying HE. P_1 then sends encryption of v to P_0 who then decrypts it using his secret key sk_0. Note that (v, r) forms an additive sharing of the desired value: $[\delta_a]_0[\delta_b]_1 + [\delta_a]_1[\delta_b]_0 = \mathsf{v} + r$.

A more detailed description for instantiating setupMULT using OT and HE is provided in the full version [83].

3.1.4 Multi-Input Multiplication Gates

3-Input Multiplication Gate. We show how to compute a 3-input multiplication gate (MULT3) with three inputs $\mathsf{a}, \mathsf{b}, \mathsf{c}$ with each input being $\langle\cdot\rangle$-shared. Similar to 2-input multiplication, we can write

$$\begin{aligned}\Delta_y = \mathsf{abc} + \delta_y &= (\Delta_a - \delta_a)(\Delta_b - \delta_b)(\Delta_c - \delta_c) + \delta_y \\ &= \Delta_{abc} - \Delta_{ab}\delta_c - \Delta_{bc}\delta_a - \Delta_{ac}\delta_b + \Delta_a\delta_{bc} + \Delta_b\delta_{ac} \\ &\quad + \Delta_c\delta_{ab} - \delta_{abc} + \delta_y.\end{aligned}$$

Here we need to generate the $[\cdot]$-sharing of four terms, namely $\delta_{ab}, \delta_{bc}, \delta_{ac}$ and δ_{abc} which is done by protocol setupMULT3. The protocol can be instantiated using either OT or HE in a similar fashion to that of setupMULT and the details are provided in the full version [83].

Multi-Input Multiplication Gate. We can extend our method to handle a 4-input multiplication (MULT4) gate and in the most general case, an N-input multiplication gate (MULTN) for any positive constant N, without inflating the online communication which remains just 2 ring elements independent of the fan-in of the gate. In contrast, the previous solution [78] requires an online communication of $2N$ ring elements for an N-input multiplication gate. Note that our improved online communication comes at the cost of an expensive setup and hence to maintain balance, we use $N \in \{3,4\}$ in our applications. We provide more details of [78] along with a comparison to our protocol in §A.3.

A more detailed description of MULT3, MULT4 and MULTN is given in the full version [83].

3.2 2PC in Boolean World

All the protocols mentioned above work over a Boolean ring ($\mathbb{Z}_{2^1}$) as well. This can be achieved by replacing additions (or subtractions) with XORs and multiplications with ANDs.

Negation Protocol. Given the **B**-sharing of a bit u as $\langle\mathsf{u}\rangle^{\mathbf{B}} = ([\delta_u], \Delta_u)$, the goal of a NOT protocol is to generate the boolean sharing of $\bar{\mathsf{u}}$. This can be done locally by setting $\Delta_{\bar{u}} = 1 \oplus \Delta_u$ and $[\delta_{\bar{u}}] = [\delta_u]$.

3.3 2PC in Yao World

For the Yao world, we follow the sharing semantics introduced by ABY [39]. For a wire u with value $\mathsf{v} \in \{0,1\}$, party P_0 acts as the garbler with the zero-key on the wire (K_u^0) being its share, while P_1 acts as the evaluator with the actual key (K_u^v) as its share. More formally, $\langle\mathsf{v}\rangle_0 = \mathsf{K}_u^0$ and $\langle\mathsf{v}\rangle_1 = \mathsf{K}_u^v$.

We use the free-XOR technique [66] in the garbling scheme, which enables the XOR gates to be evaluated without any communication. Here, the one-key for a wire is defined as a fixed offset from the zero-key as $\mathsf{K}_u^1 = \mathsf{K}_u^0 \oplus R$ with the least significant bit (LSB) of value R being set to 1 to enable

point-and-permute [10]. The value R is chosen by P_0 and is fixed across all the wires in the circuit.

To generate a $\langle\cdot\rangle$-sharing of a bit v, protocol $\mathsf{SHARE}(P_i,\mathsf{v})$ proceeds as follows: P_0 chooses a random zero-key $\mathsf{K}_\mathsf{u}^0 \in_R \{0,1\}^\kappa$ and sets $\mathsf{K}_\mathsf{u}^1 = \mathsf{K}_\mathsf{u}^0 \oplus R$, where κ denotes the computational security parameter. If $P_i = P_0$, P_0 sends $\mathsf{K}_\mathsf{u}^\mathsf{v}$ to P_1. For the case when $P_i = P_1$, parties engage in a cOT_κ^1 with P_0 being the sender and P_1 being the receiver. Here P_0 inputs the correlation function $f_R(x) = x \oplus R$ and obtains $(\mathsf{K}_\mathsf{u}^0, \mathsf{K}_\mathsf{u}^1 = \mathsf{K}_\mathsf{u}^0 \oplus R)$ while P_1 inputs v as choice bit and receives $\mathsf{K}_\mathsf{u}^\mathsf{v}$ as the output.

To generate a $\langle\cdot\rangle$-sharing of an ℓ-bit value v, parties execute the $\mathsf{SHARE}()$ protocol on each of its bits ($\mathsf{v}[j]$ for $j \in \{0,\ell-1\}$) in parallel. For a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, we abuse the notation slightly and use $\langle\mathsf{v}\rangle$ to denote the $\langle\cdot\rangle$-sharing corresponding to each bit of v. We refer readers to ABY [39] for a formal description of the two-party Yao world and the operations.

4 Mixed Protocol Conversions

In this section, we show techniques to convert the shared values among the three protocols, namely– Arithmetic, Boolean, and Yao. We use the superscripts $\{\mathbf{A},\mathbf{B},\mathbf{Y}\}$ to distinguish the sharing and the respective protocols in the Arithmetic, Boolean, and Yao world respectively.

4.1 Standard Conversions

Here we detail the conversions amongst the three protocols. While most of the conversions of ABY [39] demand the execution of OT in the online phase, our protocols invoke OT in the setup phase only. This makes the online phase of the conversions– (a) free of any cryptographic operations and (b) run for just 1 round as opposed to 2 rounds for OT in ABY (cf. Tab. 2), except the Arithmetic to Boolean conversion.

Y2B. Given the $\langle\cdot\rangle^\mathbf{Y}$-sharing of a bit $\mathsf{u} \in \{0,1\}$, the goal is to generate its equivalent Boolean sharing. As observed in ABY, since the last bit of the zero and one key are distinct, XORing the LSB of K_u^0 and $\mathsf{K}_\mathsf{u}^\mathsf{u}$ results in the underlying bit u. Hence, each P_i for $i \in \{0,1\}$ Boolean-shares the LSB of their respective shares $\langle\mathsf{u}\rangle_i^\mathbf{Y}$ followed by locally XORing the shares to obtain the desired result. We note that P_0 can perform $\mathsf{SHARE}^\mathbf{B}(P_0,\mathsf{LSB}(\mathsf{K}_\mathsf{u}^0))$ already in the setup phase.

B2Y. To convert $\langle\mathsf{u}\rangle^\mathbf{B}$ to its equivalent $\langle\cdot\rangle^\mathbf{Y}$-sharing, P_i for $i \in \{0,1\}$ first locally sets $\mathsf{u}_i = (1-i)\cdot\Delta_\mathsf{u} \oplus [\delta_\mathsf{u}]_i$. It is easy to verify that $\mathsf{u} = \mathsf{u}_0 \oplus \mathsf{u}_1$. This is followed by party P_i generating $\langle\mathsf{u}_i\rangle^\mathbf{Y}$ by executing the $\mathsf{SHARE}^\mathbf{Y}(P_i,\mathsf{u}_i)$ protocol as described in §3.3. Given $\langle\mathsf{u}_0\rangle^\mathbf{Y}, \langle\mathsf{u}_1\rangle^\mathbf{Y}$, the parties can locally compute $\langle\mathsf{u}\rangle^\mathbf{Y} = \langle\mathsf{u}_0\rangle^\mathbf{Y} \oplus \langle\mathsf{u}_1\rangle^\mathbf{Y}$ using the free-XOR technique [66]. In our solution, we observe that parties can generate $\langle\mathsf{u}_1\rangle^\mathbf{Y}$ in the setup phase, with u_1 available in the setup phase itself. This observation allows us to shift the OT run to the setup phase, as opposed to ABY [39].

A2Y. The conversion from $\langle\mathsf{v}\rangle^\mathbf{A}$ to its equivalent $\langle\cdot\rangle^\mathbf{Y}$-sharing proceeds similar to that of the B2Y conversion. Party P_i for $i \in \{0,1\}$ locally sets $\mathsf{v}_i = (1-i)\cdot\Delta_\mathsf{v} - [\delta_\mathsf{v}]_i$ so that $\mathsf{v} = \mathsf{v}_0 + \mathsf{v}_1$. During the setup phase, P_0 garbles a two-input adder circuit which computes $\mathsf{y} = \mathsf{x}_0 + \mathsf{x}_1$, given the inputs $\mathsf{x}_0, \mathsf{x}_1 \in \mathbb{Z}_{2^\ell}$. The garbled circuit is then sent to P_1. In parallel, parties execute $\mathsf{SHARE}^\mathbf{Y}(P_1,\mathsf{v}_1)$ to generate $\langle\mathsf{v}_1\rangle^\mathbf{Y}$. During the online phase, parties execute $\mathsf{SHARE}^\mathbf{Y}(P_0,\mathsf{v}_0)$ to generate $\langle\mathsf{v}_0\rangle^\mathbf{Y}$. This is followed by P_1 locally evaluating the garbled adder circuit to generate $\langle\mathsf{v}\rangle^\mathbf{Y}$ which is our desired result. The adder circuit consists of ℓ AND gates [20]. Using the half-gates technique [99], this has setup communication of $2\ell\kappa$ bits.

Y2A. To convert $\langle\mathsf{v}\rangle^\mathbf{Y}$ to $\langle\mathsf{v}\rangle^\mathbf{A}$, parties proceed similarly to ABY [39] as follows: During the setup phase, P_0 samples a random value $r \in_R \mathbb{Z}_{2^\ell}$ and executes $\mathsf{SHARE}^\mathbf{Y}(P_0,r)$ and $\mathsf{SHARE}^\mathbf{A}(P_0,r)$ to generate $\langle r\rangle^\mathbf{Y}$ and $\langle r\rangle^\mathbf{A}$ respectively. In parallel, P_0 garbles an Adder circuit and sends the garbled circuit along with the decoding information to P_1. During the online phase, P_1 evaluates the garbled circuit with inputs $\langle\mathsf{v}\rangle^\mathbf{Y}$ and $\langle r\rangle^\mathbf{Y}$ to generate $\langle\mathsf{v}+r\rangle^\mathbf{Y}$. Using the decoding information, P_1 obtains the value $(\mathsf{v}+r)$ in clear followed by executing $\mathsf{SHARE}^\mathbf{A}(P_1,\mathsf{v}+r)$ to generate $\langle\mathsf{v}+r\rangle^\mathbf{A}$. Parties then locally compute $\langle\mathsf{v}\rangle^\mathbf{A} = \langle\mathsf{v}+r\rangle^\mathbf{A} - \langle r\rangle^\mathbf{A}$.

A2B. To convert an arithmetic share $\langle\mathsf{v}\rangle^\mathbf{A}$ to its equivalent Boolean share, parties use a Boolean Adder circuit similar to that of the A2Y conversion. Here, party P_i for $i \in \{0,1\}$ locally sets $\mathsf{v}_i = (1-i)\cdot\Delta_\mathsf{v} - [\delta_\mathsf{v}]_i$ followed by executing $\mathsf{SHARE}^\mathbf{B}(P_i,\mathsf{v}_i)$ to generate $\langle\mathsf{v}_i\rangle^\mathbf{B}$. Parties then evaluate the circuit using the 2PC protocol as described in §3. As mentioned in ABY [39] and ABY3 [73], the adder circuit can either be instantiated in its size-optimized [20] or depth-optimized variant (Parallel-prefix Adder [69]) and both these methods result in a non-constant (dependent on ℓ) number of rounds. A constant-round solution is to use $\mathsf{Y2B}(\mathsf{A2Y}(\langle\mathsf{v}\rangle^\mathbf{A}))$.

Bit2A. Here the goal is to generate the arithmetic sharing of a bit $\mathsf{v} \in \{0,1\}$, given its Boolean sharing $\langle\mathsf{v}\rangle^\mathbf{B}$. Let v^a denote the value of bit v when viewed over an ℓ-bit ring. Then for $\mathsf{v} = \mathsf{v}_0 \oplus \mathsf{v}_1$, we can write $\mathsf{v}^\mathsf{a} = \mathsf{v}_0^\mathsf{a} + \mathsf{v}_1^\mathsf{a} - 2\mathsf{v}_0^\mathsf{a}\mathsf{v}_1^\mathsf{a}$. We make use of this observation in the rest of the paper several times. Note that $\mathsf{v}^\mathsf{a} = (\Delta_\mathsf{v} \oplus \delta_\mathsf{v})^\mathsf{a} = \Delta_\mathsf{v}^\mathsf{a} + \delta_\mathsf{v}^\mathsf{a} - 2\Delta_\mathsf{v}^\mathsf{a}\delta_\mathsf{v}^\mathsf{a}$.

During the setup phase, parties interactively generate the $[\cdot]$ sharing of value $\delta_\mathsf{v}^\mathsf{a}$. During the online phase, P_i for $i \in \{0,1\}$ locally computes $[\mathsf{v}^\mathsf{a}]_i = i\cdot\Delta_\mathsf{v}^\mathsf{a} + (1-2\Delta_\mathsf{v}^\mathsf{a})\cdot[\delta_\mathsf{v}^\mathsf{a}]_i$ and executes $\mathsf{SHARE}^\mathbf{A}(P_i,[\mathsf{v}^\mathsf{a}]_i)$ to generate $\langle[\mathsf{v}^\mathsf{a}]_i\rangle^\mathbf{A}$. This is followed by parties locally computing $\langle\mathsf{v}^\mathsf{a}\rangle^\mathbf{A} = \langle[\mathsf{v}^\mathsf{a}]_0\rangle^\mathbf{A} + \langle[\mathsf{v}^\mathsf{a}]_1\rangle^\mathbf{A}$.

Now we describe how to generate $[\delta_\mathsf{v}^\mathsf{a}]$ in the setup phase, given the $[\cdot]$-sharing of bit δ_v. Since $\delta_\mathsf{v} = [\delta_\mathsf{v}]_0 \oplus [\delta_\mathsf{v}]_1$, we can write $\delta_\mathsf{v}^\mathsf{a} = [\delta_\mathsf{v}^\mathsf{a}]_0 + [\delta_\mathsf{v}^\mathsf{a}]_1 - 2([\delta_\mathsf{v}^\mathsf{a}]_0[\delta_\mathsf{v}^\mathsf{a}]_1)$. The parties first execute cOT_ℓ^1 with P_0 as sender and P_1 as receiver. P_0 inputs the correlation $f_j(x) = x + [\delta_\mathsf{v}]_0^\mathsf{a}$ and obtains $(s_0 = r, s_1 = r + [\delta_\mathsf{v}]_0^\mathsf{a})$. P_1 inputs the choice bit as $[\delta_\mathsf{v}]_1$ and obtains $s_{[\delta_\mathsf{v}]_1} = r + [\delta_\mathsf{v}]_1\cdot[\delta_\mathsf{v}]_0^\mathsf{a}$ as the output. P_0 locally sets $[([\delta_\mathsf{v}]_0^\mathsf{a}[\delta_\mathsf{v}]_1^\mathsf{a})]_0 =$

$-r$ while P_1 sets $[([\delta_v]_0^a\,[\delta_v]_1^a)]_0 = s_{[\delta_v]_1}$. Party P_i for $i \in \{0,1\}$ locally sets the $[\cdot]$-share of $[\delta_v^a]$ as $[\delta_v^a]_i = (1-i)\cdot[\delta_v]_0^a + i\cdot[\delta_v]_1^a - 2\,[([\delta_v]_0^a\,[\delta_v]_1^a)]_i$.

B2A. To convert a value $v \in \mathbb{Z}_{2^\ell}$ from its $\langle\cdot\rangle^{\mathbf{B}}$-sharing to its equivalent arithmetic sharing $\langle v\rangle^{\mathbf{A}}$, one simple solution is to follow steps similar to the Y2A conversion. Here, parties evaluate a Boolean subtraction circuit with $\langle v\rangle^{\mathbf{B}}$ and $\langle r\rangle^{\mathbf{B}}$ as the inputs, where r denotes a random value chosen by P_0. In addition, P_0 executes $\mathsf{SHARE}^{\mathbf{A}}(P_0, r)$ to generate $\langle r\rangle^{\mathbf{A}}$ as well. After the evaluation, the value $(v-r)$ is reconstructed to P_1, who further generates $\langle v-r\rangle^{\mathbf{A}}$. Parties then locally compute $\langle v\rangle^{\mathbf{A}} = \langle v+r\rangle^{\mathbf{A}} - \langle r\rangle^{\mathbf{A}}$.

As the above solution results in a non-constant round protocol in the online phase, we propose a *novel* round efficient variant which makes use of the Bit2A protocol. Our protocol was inspired from [31] that proposed a similar solution for the four party honest majority case. Here we make use of the fact that $v = \sum_{j=0}^{\ell-1} 2^j \cdot v[j]$ where $v[j]$ denotes the j^{th} bit of v. Since the parties possess $\langle v[j]\rangle^{\mathbf{B}}$ for each $j \in [0,\ell)$, they execute Bit2A conversion on $\langle v[j]\rangle^{\mathbf{B}}$ to generate its arithmetic equivalent $\langle v[j]\rangle^{\mathbf{A}}$. This results in a communication corresponding to ℓ instances of Bit2A conversions.

We observe that the online cost can be brought down to just 2 ring elements using the following approach. For each bit $v[j]$, parties locally compute the $[\cdot]$-sharing corresponding to $(v[j])^a$ as mentioned in Bit2A. Now, instead of generating the $\langle\cdot\rangle^{\mathbf{A}}$-share corresponding to each bit, P_i for $i \in \{0,1\}$ locally computes $[v]_i = \sum_{j=0}^{\ell-1} 2^j \cdot [(v[j])^a]_i$ and executes $\mathsf{SHARE}^{\mathbf{A}}(P_i, [v]_i)$ to generate $\langle [v]_i\rangle^{\mathbf{A}}$. Both parties then locally compute $\langle v\rangle^{\mathbf{A}} = \langle [v_0]\rangle^{\mathbf{A}} + \langle [v_1]\rangle^{\mathbf{A}}$. It is easy to verify that $v = [v]_0 + [v]_1$.

4.2 Special Conversions

For the three special conversions described below, the inputs are either Boolean shares or a mix of Boolean and arithmetic shares. The goal is to compute the equivalent arithmetic sharing of the product of the inputs. These conversions use the techniques of the Bit2A protocol (§4.1).

a) Protocol $\mathsf{PQ}(\langle p\rangle^{\mathbf{B}}, \langle q\rangle^{\mathbf{B}}) : \langle p\rangle^{\mathbf{B}}\langle q\rangle^{\mathbf{B}} \to \langle pq\rangle^{\mathbf{A}}$

Prep: $[\delta_p^a], [\delta_q^a], [\delta_p^a\delta_q^a]$

$(pq)^a = (\Delta_p^a + (1-2\Delta_p^a)\delta_p^a)(\Delta_q^a + (1-2\Delta_q^a)\delta_q^a)$

b) Protocol $\mathsf{PV}(\langle p\rangle^{\mathbf{B}}, \langle v\rangle^{\mathbf{A}}) : \langle p\rangle^{\mathbf{B}}\langle v\rangle^{\mathbf{A}} \to \langle pv\rangle^{\mathbf{A}}$

Prep: $[\delta_p^a], [\delta_p^a\delta_v]$

$(pv)^a = (\Delta_p^a + (1-2\Delta_p^a)\delta_p^a)(\Delta_v - \delta_v)$

c) Protocol $\mathsf{PQV}(\langle p\rangle^{\mathbf{B}}, \langle q\rangle^{\mathbf{B}}, \langle v\rangle^{\mathbf{A}}) : \langle p\rangle^{\mathbf{B}}\langle q\rangle^{\mathbf{B}}\langle v\rangle^{\mathbf{A}} \to \langle pqv\rangle^{\mathbf{A}}$

Prep: $[\delta_p^a], [\delta_q^a], [\delta_p^a\delta_q^a], [\delta_p^a\delta_v], [\delta_q^a\delta_v], [\delta_p^a\delta_q^a\delta_v]$

$(pqv)^a = (\Delta_p^a + (1-2\Delta_p^a)\delta_p^a)(\Delta_q^a + (1-2\Delta_q^a)\delta_q^a)(\Delta_v - \delta_v)$

During the online phase, parties locally generate a $[\cdot]$-sharing of the value to be computed followed by executing the $\mathsf{SHARE}^{\mathbf{A}}$ protocol on it to generate its equivalent arithmetic sharing. Then, parties locally add the resulting arithmetic shares to obtain the final result. The difference lies in the setup required for each of the conversions. The expression provided above shows the desired result in terms of corresponding Δ and δ values and the data (labelled as Prep) to be prepared in the setup phase.

As observed in the Bit2A protocol, the online phase of all these conversions consists of both parties executing arithmetic sharing of a single element resulting in one round with a communication of just 2 ring elements. We provide a detailed description of the conversions in the full version [83].

5 Building Blocks for Applications

In this section, we provide details for our building blocks that form the core of the applications that we explore in §6. We provide the formal details and communication cost analysis in the full version [83].

5.1 Scalar Product

Given the arithmetic sharing of n-element vectors $\vec{a}, \vec{b}$, the goal is to generate $\langle y\rangle^{\mathbf{A}}$ where $y = \vec{a} \odot \vec{b} = \sum_{j=1}^{n} a_i b_i$. One trivial way is to invoke the multiplication protocol from §3.1.3 corresponding to each of the n underlying multiplications. This would result in online communication linear in the vector size n. We now show how to make the online communication *independent* of the vector size.

The parties first execute the preprocessing corresponding to each of the n multiplications in parallel. Here we observe that there is no need to sample the shares of $[\delta_{y_j}]$ corresponding to each of the underlying multiplications. Instead, the parties locally sample the shares of $[\delta_y]$. During the online phase, parties first locally compute the $[\cdot]$-sharing of value Δ_{y_j} where y_j denotes $a_j b_j$. P_i for $i \in \{0,1\}$ now locally computes $[\Delta_y]_i = \sum_{j=1}^{n} [\Delta_{y_j}]_i$. This is followed by the parties mutually exchanging $[\Delta_y]$-shares to reconstruct Δ_y.

Compared with the state-of-the-art 2PC solutions in ABY [39] which require communication of 4n elements in the online phase, our protocol requires an online communication of just 2 ring elements.

5.2 Matrix Multiplication

Here we provide the details for extending our 2PC multiplication (§3.1.3) to the matrix setting. We abuse the notation slightly and use '$+$' for addition of matrices and '$-$' for subtraction. Also, we follow the $\langle\cdot\rangle$-sharing semantics for matrices as well. For $\mathbf{X}^{m\times n}$, we have $\Delta_{\mathbf{X}} = \mathbf{X} + [\delta_{\mathbf{X}}]_0 + [\delta_{\mathbf{X}}]_1$. Here $\Delta_{\mathbf{X}}, [\delta_{\mathbf{X}}]_0$ and $[\delta_{\mathbf{X}}]_1$ are matrices with dimension $m \times n$ and $x_{i,j}$ denote the $[i:j]$-th entry of $\mathbf{X}$.

Given $\mathbf{A}^{p\times q}, \mathbf{B}^{q\times r}$, protocol MATMULT proceeds as follows: During the setup phase, for $i \in [p], j \in [q], k \in$

$[r]$, parties execute $\mathsf{setupMULT}(\left[\delta_{\mathsf{a}_{i,j}}\right], \left[\delta_{\mathsf{b}_{j,k}}\right])$ to generate $\left[\delta_{\mathsf{a}_{i,j}\mathsf{b}_{j,k}}\right]$. This results in a $[\cdot]$-sharing of $\gamma_{\mathbf{AB}} = \delta_{\mathbf{A}} \circ \delta_{\mathbf{B}}$ among P_0, P_1. During the online phase, parties locally compute a $[\cdot]$-sharing of $\Delta_{\mathbf{C}}$ using the following relation:

$$\begin{aligned}\Delta_{\mathbf{C}} &= \mathbf{C} + \delta_{\mathbf{C}} = \mathbf{A} \circ \mathbf{B} + \delta_{\mathbf{C}} = (\Delta_{\mathbf{A}} - \delta_{\mathbf{A}}) \circ (\Delta_{\mathbf{B}} - \delta_{\mathbf{B}}) + \delta_{\mathbf{C}} \\ &= \Delta_{\mathbf{A}} \circ \Delta_{\mathbf{B}} - \Delta_{\mathbf{A}} \circ \delta_{\mathbf{B}} - \delta_{\mathbf{A}} \circ \Delta_{\mathbf{B}} + \gamma_{\mathbf{AB}} + \delta_{\mathbf{C}}.\end{aligned}$$

Finally, parties mutually exchange $[\Delta_{\mathbf{C}}]$ and obtain $\Delta_{\mathbf{C}}$ completing the protocol. Our protocol improved the online communication from $\mathrm{O}(pqr)$ to $\mathrm{O}(pr)$ ring elements, eliminating the dependency on dimension q.

5.3 Depth-Optimized Circuits

Parallel-prefix Adders (PPA) offer a depth-optimized solution to the binary addition between two ℓ-bit binary numbers. The best-known PPAs have $\log_2(\ell)$ depth [47]. Using ideas from [7,47], we design a PPA using two, three, and four input AND gates combined and obtain depth-optimized PPAs. For a 64-bit ring, we achieve a $2\times$ improvement in depth over existing designs along with a reduction in online communication.

Circuit	ℓ	#AND2		#AND3	#AND4	Depth	
Adder	8	15	(24)	6	1	2	(3)
BitExt	8	7	(14)	4	1	2	(3)
Adder	64	216	(384)	184	179	3	(6)
BitExt	64	41	(126)	27	47	3	(6)

Table 5: Depth-optimized Circuits for ℓ-bit rings. Previous circuits from ABY3 [73] are given in brackets.

As shown in [73], the PPA circuit can be optimized to obtain just the most significant bit (MSB), which we denote as Bit Extraction (BitExt) circuits. The efficiency gain in our PPA construction extends to BitExt circuits as well. Tab. 5 provides a summary of the results.

5.4 Comparison

As pointed out in [30,73], checking $\mathsf{x} < \mathsf{y}$ in the Fixed-Point Arithmetic (FPA) representation is equivalent to checking the sign of $\mathsf{v} = \mathsf{x} - \mathsf{y}$, which is stored in the MSB position of v.

The corresponding protocol LT begins with parties locally computing $\langle \mathsf{v} \rangle = \langle \mathsf{x} \rangle - \langle \mathsf{y} \rangle$. Let $\mathsf{v} = \mathsf{a} + \mathsf{b}$ where $\mathsf{a} = -[\delta_{\mathsf{v}}]_0$ and $\mathsf{b} = \Delta_{\mathsf{v}} - [\delta_{\mathsf{v}}]_1$. P_0, P_1 execute $\mathsf{SHARE}^{\mathbf{B}}$ on a, b respectively to generate its equivalent boolean sharing. The parties then use the Bit Extraction (BitExt, §5.3) circuit to compute $\mathsf{MSB}(\mathsf{v})$ in the boolean sharing format.

5.5 Truncation

In Fixed-Point Arithmetic (FPA), repeated multiplications result in an overflow with the fractional part doubling up in size after each multiplication. The naive solution of choosing a large enough ring to avoid the overflow is impractical for ML algorithms where the number of sequential multiplications is large. To tackle this, truncation [31,73,75] is used where the result of the multiplication is brought back to the FPA representation by chopping off the last x bits.

Below we explain how to perform truncation without affecting the communication cost for the multiplication. Our protocol is inspired by SecureML [75] and works as follows: During the online phase of multiplication, the parties first locally compute $[\mathsf{y}]$ directly instead of $[\Delta_{\mathsf{y}}]$. This is possible since $[\mathsf{y}] = [\Delta_{\mathsf{y}}] - [\delta_{\mathsf{y}}]$. Now each party locally truncates $[\mathsf{y}]$ to obtain the truncated value denoted by $[\mathsf{y}^t]$. This is followed by parties executing the $\mathsf{SHARE}^{\mathbf{A}}$ protocol on $[\mathsf{y}^t]$ to generate its arithmetic sharing. Finally, the parties locally compute $\langle \mathsf{y}^t \rangle^{\mathbf{A}} = \langle [\mathsf{y}^t]_0 \rangle^{\mathbf{A}} + \langle [\mathsf{y}^t]_1 \rangle^{\mathbf{A}}$. The correctness of the method follows trivially from SecureML.

5.6 MAX2 / MIN2

The MAX2 protocol is used to compute the maximum among two values a, b in a secure manner given $\langle \mathsf{a} \rangle^{\mathbf{A}}$ and $\langle \mathsf{b} \rangle^{\mathbf{A}}$. For this, the parties execute the LT protocol from §5.4 on $\langle \mathsf{a} \rangle^{\mathbf{A}}, \langle \mathsf{b} \rangle^{\mathbf{A}}$ to obtain $\langle \mathsf{u} \rangle^{\mathbf{B}} = \langle a < b \rangle^{\mathbf{B}}$. Note that $\mathsf{MAX2}(\mathsf{a}, \mathsf{b}) = \mathsf{u} \cdot (\mathsf{b} - \mathsf{a}) + \mathsf{a}$. Hence, parties can use the PV protocol from §4.2 to compute the desired result. The MIN2 protocol proceeds similarly except that $\mathsf{MIN2}(\mathsf{a}, \mathsf{b}) = \mathsf{u} \cdot (\mathsf{a} - \mathsf{b}) + \mathsf{b}$.

5.7 MAX3 / MIN3

Given the arithmetic sharing $\langle \mathsf{a} \rangle^{\mathbf{A}}, \langle \mathsf{b} \rangle^{\mathbf{A}}, \langle \mathsf{c} \rangle^{\mathbf{A}}$, the goal of the MAX3 protocol is to find the maximum value among the three. For this, we optimize the solution proposed by [78] which results in an improvement of $24.5\times$ in terms of the communication and $1.3\times$ in rounds in the online phase. The parties first securely compare the pairs $(\mathsf{a}, \mathsf{b}), (\mathsf{a}, \mathsf{c})$ and (b, c) using the LT protocol from §5.4 and obtain $\langle \mathsf{u}_1 \rangle^{\mathbf{B}}, \langle \mathsf{u}_2 \rangle^{\mathbf{B}}$ and $\langle \mathsf{u}_3 \rangle^{\mathbf{B}}$ respectively. Here $\mathsf{u}_1 = 1$ if $\mathsf{a} < \mathsf{b}$ and 0 otherwise. u_2 and u_3 are defined likewise . Now the maximum among the three, denoted by y, can be written as $\mathsf{y} = \overline{\mathsf{u}_1} \cdot \overline{\mathsf{u}_2} \cdot \mathsf{a} + \mathsf{u}_1 \cdot \overline{\mathsf{u}_3} \cdot \mathsf{b} + \mathsf{u}_2 \cdot \mathsf{u}_3 \cdot \mathsf{c}$.

Given $\langle \mathsf{u}_1 \rangle^{\mathbf{B}}, \langle \mathsf{u}_2 \rangle^{\mathbf{B}}, \langle \mathsf{u}_3 \rangle^{\mathbf{B}}$ and $\langle \mathsf{a} \rangle^{\mathbf{A}}, \langle \mathsf{b} \rangle^{\mathbf{A}}, \langle \mathsf{c} \rangle^{\mathbf{A}}$, the parties can use the PQV protocol from §4.2 to obtain each term in the expression for y and can locally add them to obtain the desired result. As an optimization, we can combine the online phase corresponding to all three executions of the PQV protocol into one. This reduces the online communication from six to two ring elements.

The protocol for MIN3, which computes the minimum among the three values can be obtained by slightly modifying the protocol for MAX3. The difference lies in the expression for computing the minimum which will now be $\mathsf{y} = \mathsf{u}_1 \cdot \mathsf{u}_2 \cdot \mathsf{a} + \overline{\mathsf{u}_1} \cdot \mathsf{u}_3 \cdot \mathsf{b} + \overline{\mathsf{u}_2} \cdot \overline{\mathsf{u}_3} \cdot \mathsf{c}$.

We observe that the protocol described above can be modified slightly to compute the index of the maximum (or minimum) among a set of three values. We use

ArgMax/ArgMin to denote such a protocol and the details are given in the full version [83].

5.8 Non-linear Activation Functions

We show how to compute two of the most widely used non-linear activation functions for PPML: ReLU and Sigmoid. ReLU function, defined as $\mathsf{ReLU}(v) = \max(0, v)$, can be written as $\mathsf{ReLU}(v) = \bar{u}v$, where $u = 1$ if $v < 0$ and 0 otherwise. To compute this, parties first execute the LT protocol from §5.4 on v to obtain $\langle u \rangle^{\mathbf{B}}$ followed by executing the PV protocol from §4.2 on $\langle \bar{u} \rangle^{\mathbf{B}}$ and $\langle v \rangle^{\mathbf{A}}$ to obtain the desired result. For Sigmoid, we use the MPC-friendly version [30, 73, 75] defined as $\mathsf{Sig}(v) = \overline{u_1} u_2 (v + 1/2) + \overline{u_2}$, where $u_1 = 1$ if $v + 1/2 < 0$ and $u_2 = 1$ if $v - 1/2 < 0$.

5.9 Maxpool and Minpool

Given the arithmetic sharing of an n-element vector $\vec{x} = (x_1, \ldots, x_n)$ of values with $x_j \in \mathbb{Z}_{2^\ell}$ for $j \in \{1, \ldots, n\}$, the goal of the Maxpool protocol is to compute the arithmetic sharing of the maximum value among the n values.

For this, parties arrange the n values into an N-ary tree (tournament) composed of MAXN blocks with depth $\log_N(n)$ and evaluate in a top-down fashion [64]. In the recent work of [78], a maxpool using MAX3 was proposed where three values are compared at a time. In this work, we use our optimized MAX3 protocol from §5.7 as the building block for computing Maxpool. The improvement in rounds as well as communication of our MAX3 protocol over [78] directly translates to this case as well. We provide an empirical comparison for the Maxpool protocol in §6.1. Using MIN3 instead of MAX3 will directly provide a solution for Minpool, where the goal is to find the minimum among the values.

5.10 Equality Testing

Given $\langle a \rangle^{\mathbf{A}}, \langle b \rangle^{\mathbf{A}}$, the goal of the Equality Testing (EQ) protocol is to check whether $a \stackrel{?}{=} b$ or not. An equivalent formulation of the problem [18, 78] is to check if all the bits of $a - b$ are 0 or not. This simple primitive is crucial in building efficient protocol for applications like Circuit-based Private Set Intersection [85, 87, 88] (cf. §6.3), the Table Lookup Protocol from [40], and Data Mining [18].

We begin with the observation that if $x = y$, then using our sharing semantics we can write $\Delta_x - [\delta_x]_0 - [\delta_x]_1 = \Delta_y - [\delta_y]_0 - [\delta_y]_1$. Assuming $v_0 = (\Delta_x - [\delta_x]_0) - (\Delta_y - [\delta_y]_0)$ and $v_1 = [\delta_x]_1 - [\delta_y]_1$, the problem now reduces to checking whether $v_0 \stackrel{?}{=} v_1$ or not. Note that the value v_i can be locally computed by party P_i for $i \in \{0, 1\}$.

Protocol EQ proceeds as follows: P_i for $i \in \{0, 1\}$ locally computes v_i and executes $\mathsf{SHARE}^{\mathbf{B}}$ to generate $\langle v_i \rangle^{\mathbf{B}}$. The parties then compute $\langle v \rangle^{\mathbf{B}} = \mathsf{NOT}(\langle v_0 \rangle^{\mathbf{B}} \oplus \langle v_1 \rangle^{\mathbf{B}})$. Note that checking $v_0 = v_1$ is the same as checking whether all the bits of v are 1 or not. For this, the parties use AND4 gates and a tree structure, where 4 bits are taken at a time and the AND of them is computed in one go. This approach improves the round complexity by a factor of 2 over the traditional approach using AND2 gates. In concrete terms for a 64 bit ring, our solution improves over the protocol of [18] by $2\times$ in online rounds and by $2.4\times$ in online communication.

6 Applications and Benchmarks

All secure two-party applications using Boolean sharing (**B**) or Arithmetic sharing (**A**) directly benefit from our improvement in the online phase of our protocols. In this section, we give four applications with further improvements: i) AES which benefits from AND3 gates (§6.2), ii) Circuit-based Private Set Intersection (PSI) which benefits from our improved Equality Tests (§6.3), ii) Biometric Matching which benefits from our new *dimension-independent* Scalar Product and Minpool protocols (§6.4), and iv) Privacy-Preserving Machine Learning (PPML), specifically training and inference of Logistic Regression and Neural Networks which benefit from many of our improved protocol building blocks (§6.5). Since Maxpool/Minpool is an essential building block for several applications like K-means clustering [25], face-recognition [93], and fingerprint-matching [15, 43], we provide a separate analysis for Maxpool in §6.1.

To showcase the practicality of our constructions, we have implemented our protocols and compare them with their closest competitors. We implemented our protocols using the ENCRYPTO library [41] in C++17 over a 64-bit ring. Each experiment is run 15 times and the average values are reported. The benchmarking is performed over a LAN of 25Gbps bandwidth and a WAN of 75Mbps bandwidth. Over the LAN, we use two machines, each equipped with a 3.5 GHz Intel (R) Xeon (R) Gold 6144 CPU and 64 GB of RAM. The WAN was instantiated using n1-standard-8 instances of Google Cloud[1] with machines located in East Australia (P_0) and South East Asia (P_1). Over the WAN, machines are equipped with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors supporting hyper-threading, with 8 vCPUs, and 30 GB of RAM. The average round-trip time (rtt), which was taken as the time for communicating 128 KB of data, turned out to be 0.056 ms for LAN and 60.19 ms for WAN.

6.1 Maxpool

Here we provide an empirical analysis of our Maxpool protocol from §5.9 and compare it with its competitors. We consider vectors with dimensions $n \in \{1024, 65536\}$. We have evaluated both round-optimized and communication-optimized variants of the Maxpool protocol. In the round-optimized variant proposed by SecureML [75], a garbled circuit is used to evaluate the maximum among n elements. This method requires converting Arithmetic shares to Yao shares and back, which can be tackled using A2Y and Y2A

[1] https://cloud.google.com

conversions. In the communication-optimized variant, we use the tree-based approach where either two or three elements are compared at a time as described in §5.9.

Ref.	Type	n = 1,024		n = 65,536	
		Comm [KB]	Rounds	Comm [KB]	Rounds
[75]	GC	2,056	4	131,584	4
ABY2.0	GC	1,024	**2**	65,536	**2**
[78]	MAX2	258	50	16,512	80
ABY2.0	MAX2	**53**	40	**3,408**	64
[78]	MAX3	492	35	31,679	55
ABY2.0	MAX3	63	28	4,080	44

Table 6: Online communication and rounds of Maxpool protocols. Best results in bold. n is the number of input elements.

Based on the building block used to instantiate Maxpool, the analysis can be divided into three cases – i) *Case I:* where the garbled circuit is used, ii) *Case II:* only MAX2 is used, and iii) *Case III:* a mix of MAX3 and MAX2 are used. For Case I, we compare with SecureML [75], while ours is compared with [78] for the rest. Table 6 summarizes the cost for the online phase of the Maxpool protocol. It is evident from the table that our protocols outperform [75, 78] in both communication and rounds for the online phase in all three cases.

For Case I, our round-optimized variant has a 2× improvement over SecureML [75] in both online communication and rounds. This is due to our efficient A2Y and Y2A conversions. For Case II, we improve upon [78] by a factor of 6.2× in online communication and 1.3× in rounds. Similarly, for Case III, the respective improvements over [78] are 9.6× and 1.3×. For cases II&III, while the improvement in online rounds is due to our efficient comparison protocol, improvement in communication is primarily contributed by our PQV protocol from §4.2. We also note that [78] improved the online rounds by 1.4× by switching from MAX2 to MAX3 as the building block for Maxpool at the expense of 1.9× higher online communication. In contrast, our solution improves the online rounds by 1.4× with a minimal overhead of 1.2× in online communication.

For the round-optimized variant, our protocol incurs an additional communication of just 2KB over SecureML in the setup phase. For the communication-optimized variant, we improve upon [78] for both MAX2 and MAX3 in terms of communication in the setup phase. This improvement results from our improved comparison protocol.

6.2 Improved S-box for AES

In a privacy-preserving AES [51, 86], the goal is to enable P_0 to encrypt her message x using a key k held P_1. The privacy guarantee is that P_0 gets the corresponding ciphertext while leaking nothing else. This has several applications in PSI [48, 58] and encrypted databases [2, 22]. Since the MixColumns and AddRoundKey operations can be evaluated using only free XOR gates [51], the focus was shifted to building efficient protocols for evaluating S-boxes as its core block. While [21] gives a depth-optimized S-box of 34 AND gates with an AND-Depth of 4, [19] gives a size-optimized solution with 32 AND gates and AND-Depth 6.

We give a new construction for the AES S-box that results in an effective AND-Depth of only 3. On a high level, we start with the three-layer construction of [19, 21] and optimize the middle layer (inversion layer) by replacing some of the AND2 gates with AND3 gates. This optimization is crucial since AES-128, AES-192 and AES-256 have 10, 12, and 14 sequential calls to layers of S-boxes resulting in a respective saving of 10, 12, and 14 rounds of interaction over [21]. We provide the empirical analysis in Table 7 and defer a detailed description to the full version [83].

Cipher	Ref.	#AND	Setup	Online	
			Comm [KB]	Comm [KB]	Rounds
AES 128	[21]	5,440	88.98	2.66	40
	[19]	5,120	**83.75**	2.50	60
	ABY2.0	5,440	98.13	**1.33**	**30**

Table 7: Communication and rounds for Secure evaluation of AES. Best results in bold.

In the setup phase, we used 4-OT_1^1 for AND2 gates and 8-OT_4^1 for AND3 gates. With the optimization of [40] applied, one instance of 4-OT_1^1 requires communication of 134 bits while 8-OT_4^1 takes 253 bits. Our protocol outperforms [21] and [19] in terms of both online communication and rounds.

6.3 Circuit-Based PSI

Circuit-based PSI [50] allows us to efficiently compute *variants* of the Private Set Intersection (PSI) functionality by securely evaluating a Boolean circuit. Today's most efficient protocols in this area [85, 87–89] do this by using hashing techniques and then evaluating a Boolean circuit that checks for equality among several bit strings using secure 2PC.

In fact, for today's most efficient circuit-based PSI protocol of [87], the majority of the computation, as well as communication, is spent on this two-party Equality Checking protocol. To be precise, 96% of the overall communication (cf. [87, Tab. 3]) and 34% – 63% of the overall runtime (cf. [87, Tab. 5]) is spent on Equality Checking. Plugging in our efficient Equality Checking protocol from §5.10 into the PSI protocol of [87] results in a direct improvement of $\approx 1.3\times$ in runtime and $\approx 2.4\times$ in communication in the online phase.

6.4 Biometric (Minimum Euclidean Distance)

Given a database owner with m biometric samples $(\vec{s_1}, \ldots, \vec{s_m})$ and a party with its biometric sample $\vec{c}$, the goal of privacy-preserving biometric matching is to find out the "minimum distance" of $\vec{c}$ from the database. This method is used for various traits of biometrics such as face-recognition [42, 49] and fingerprint-matching [15, 51]. Some of these works use the Squared Euclidean Distance (SED) as the metric to

compute the distance between two vectors. For two n-element vectors $\vec{a}, \vec{b}$, SED is defined as $\mathsf{SED}(\vec{a},\vec{b}) = \sum_{j=1}^{\mathsf{n}} (\mathsf{a}_j - \mathsf{b}_j)^2$. Note that for $\vec{y} = \vec{a} - \vec{b}$, $\mathsf{SED}(\vec{a},\vec{b}) = \vec{y} \odot \vec{y}$.

In our framework, P_0 is the database owner while P_1 is the party with the sample to be checked. For finding the nearest sample securely, the parties first generate an arithmetic sharing of both the database samples and the query according to our sharing semantics. Given $\langle \vec{s}_j \rangle^{\mathbf{A}}$ for $j \in \{1,\ldots,m\}$ and $\langle \vec{c} \rangle^{\mathbf{A}}$, the parties locally compute $\langle \vec{x}_j \rangle^{\mathbf{A}} = \langle \vec{s}_j \rangle^{\mathbf{A}} - \langle \vec{c} \rangle^{\mathbf{A}}$. This is followed by running the dot product protocol from §5.1 on each $\langle \vec{x}_j \rangle^{\mathbf{A}}$ with itself to generate $\langle \mathsf{y}_j \rangle^{\mathbf{A}} = \langle \vec{x}_j \odot \vec{x}_j \rangle^{\mathbf{A}}$. Note that the vector $\langle \vec{y} \rangle^{\mathbf{A}} = \{\langle \mathsf{y}_1 \rangle^{\mathbf{A}}, \ldots, \langle \mathsf{y}_m \rangle^{\mathbf{A}}\}$ represents the SED of the query with each of the database samples. To find the minimum among the elements of $\vec{y}$ given the arithmetic sharing of its elements, the parties can use either of the two methods described below.

In the first method, P_0 generates a garbled circuit that can compute the minimum among m inputs and sends this circuit to P_1. The parties then execute the A2Y conversion on each $\langle \mathsf{y}_j \rangle^{\mathbf{A}}$ for $j \in \{1,\ldots,m\}$ to generate $\langle \mathsf{y}_j \rangle^{\mathbf{Y}}$. P_1 evaluates the circuit to obtain the desired result in $\langle \cdot \rangle^{\mathbf{Y}}$-sharing. This method will result in a constant round solution, but the communication will be large. Another option is to use our Minpool protocol from §5.9 which results in a communication-efficient solution, but will require a non-constant number of rounds.

Ref.	Type	$m = 1{,}024$		$m = 4{,}096$		$m = 16{,}384$	
		Rounds	Comm [KB]	Rounds	Comm [KB]	Rounds	Comm [KB]
[39]	A+Y	5	2,312	5	9,248	5	36,992
ABY2.0	A+Y	**3**	1,040	**3**	4,160	**3**	16,640
[78]	A+B	36	748	41	3,003	46	12,014
ABY2.0	A+B	29	**51**	33	**205**	37	**818**

Table 8: Online rounds and communication of Minimum Euclidean Distance. Best results in bold. m is the number of biometric samples.

An empirical analysis for the online phase of the two aforementioned variants is given in Tab. 8. We consider databases with $m \in \{1{,}024, 4{,}096, 16{,}384\}$ samples. Each biometric sample has a dimension of $\mathsf{n} = 8$.

For the round-optimized variant, we improve upon ABY [39] by 2.2× in communication and and by 1.6× in rounds in the online phase. Similarly, for the communication-optimized variant, our improvements over [78] are 14.7× in communication and 1.3× in rounds. The overhead in the setup cost for our protocol over ABY [39] and [78] is similar to that of Maxpool (§6.1) since Minpool forms the majority of the computation for Biometric Matching.

6.5 Privacy-Preserving Machine Learning (PPML)

In the domain of PPML [30, 31, 73, 75], we show that Logistic Regression and Neural Networks can be substantially improved with our building blocks. While we chose the above applications, our building blocks are sufficient to perform training and inference of Linear Regression and Convolutional Neural Networks [31] as well as inference of Support Vector Machines [30] and Binarized Neural Networks [27].

The training phase for the aforementioned algorithms consists of two stages: (i) a *forward propagation* phase, where the model computes the output given the input; and (ii) a *backward propagation* phase, where the model parameters are adjusted according to the difference in the computed output and the actual one. The inference phase can be viewed as one pass of the forward propagation alone. In our work, we use the technique of Batching [73, 75], where the entire set of samples is divided into batches of size B and a combined update function is applied to the weight vectors.

For the training phase, we follow [31, 73] and benchmark the *number of iterations per minute* (#it/min) over both LAN and WAN. The values are reported over batch sizes of $\{128, 256, 512\}$ and with feature sizes $\mathsf{n} \in \{100, 900\}$. For the inference, we report the online runtime as well as the *throughput* (TP) for the aforementioned feature sizes. Runtime shows the impact of rounds on the overall performance, while TP denotes the numbers of queries the framework can process in a minute and allows to analyse the impact of communication.

Logistic Regression. In Logistic Regression, one iteration comprises of updation of the weight vector $\vec{\mathbf{w}}$ using the gradient descent algorithm (GD) as follows:

$$\vec{\mathbf{w}} = \vec{\mathbf{w}} - \frac{\alpha}{B} \mathbf{X}_i^T \circ (\mathsf{Sig}(\mathbf{X}_i \circ \vec{\mathbf{w}}) - \mathbf{Y}_i) .$$

Here α denotes the learning rate and $\mathbf{X}_i$ denotes a subset of batch size B, randomly selected from the entire dataset in the i-th iteration.

Batch Size	Ref.	LAN (#it/min)		WAN (#it/min)	
		$\mathsf{n} = 100$	$\mathsf{n} = 900$	$\mathsf{n} = 100$	$\mathsf{n} = 900$
128	[75]	29,112	27,273	108	104
	ABY2.0	**176,471**	**149,626**	**162**	**162**
256	[75]	25,829	24,058	107	97
	ABY2.0	**163,043**	**117,188**	**162**	**162**
512	[75]	23,292	22,247	104	83
	ABY2.0	**110,906**	**98,847**	**162**	**162**

Table 9: Comparison of the online throughput of ABY2.0 and SecureML [75] for Logistic Regression Training. Best results are in bold and larger is better. n is the number of features.

For the case of training, the data owner possesses the matrices $\mathbf{X}, \mathbf{Y}$ and the initial weights ($\vec{\mathbf{w}}$) are all set to 0. During the forward propagation, $\mathbf{X}_i \circ \vec{\mathbf{w}}$ is first computed followed by applying the sigmoid (Sig) function on it. During the backward propagation, the weight vector is updated according to the equation above. The update function requires computation of a series of matrix multiplications, which can be achieved using our dot product protocol from §5.1. The

operations of subtraction as well as multiplication by a public constant can be performed locally.

Tab. 9 gives our benchmarks for Logistic Regression training. Over SecureML [75], we have improvements in the range 4.4×-6.1× for LAN and in the range 1.5×-2.0× for WAN. The improvement stems from our round efficient comparison protocol from §5.4 that forms the building block for the activation function ReLU as well as our scalar product protocol from §5.1 that has a communication independent of the size of the vector. Note that over WAN, the throughput of our protocol remains unchanged across feature sizes as well as batch sizes. This discrepancy is due to the effect of communication on the rtt. In detail, the rtt is in the order of microseconds for LAN and scales with the communication size, whereas rtt in the WAN is in the order of milliseconds and does not scale with communication up to a threshold, within which all our protocols operate.

Parameter	Ref.	LAN		WAN	
		n = 100	n = 900	n = 100	n = 900
Runtime (ms)	[75]	1.60	1.69	496.08	504.96
	ABY2.0	**0.29**	**0.29**	**308.16**	**308.16**
Throughput (Queries/min)	[75]	5,342.61	1,193.01	16.08	3.58
	ABY2.0	**42,372.41**	**42,371.11**	**39.88**	**39.88**

Table 10: Comparison of the online runtime and throughput of ABY2.0 and SecureML [75] for Logistic Regression Inference. Best results in bold. n is the number of features.

Tab. 10 gives our benchmarks for Logistic Regression inference. We improve the online runtime over SecureML [75] by 5.5× for LAN and 1.6× for WAN, and the online throughput by 7.9×-35.5× in LAN and 2.5×-11.1× in WAN.

Neural Networks (NN). Neural Networks are stronger than regression algorithms since they can learn more complex relationships between high dimensional input and output data.

Batch Size	Ref.	LAN (#it/min)		WAN (#it/min)	
		n = 100	n = 900	n = 100	n = 900
128	[75]	3,593	3,559	17	17
	ABY2.0	**12,448**	**12,343**	**42**	**42**
256	[75]	3,578	3,521	17	17
	ABY2.0	**9,259**	**9,156**	**42**	**42**
512	[75]	3,330	3,323	15	15
	ABY2.0	**9,177**	**9,146**	**42**	**42**

Table 11: Comparison of the online throughput of ABY2.0 and SecureML [75] for NN Training. Best results in bold and larger is better. n is the number of features.

In our work, we follow previous works [30, 73, 75] and consider a Neural Network with two hidden layers, each having 128 nodes followed by an output layer of 10 nodes. We use ReLU as the activation function over the nodes. Moreover, for training we use the MPC-friendly variant of the softmax function [75] which is defined as $f(\mathsf{v}_i) = \mathsf{ReLU}(\mathsf{v}_i)/\sum_{j=1}^{m}\mathsf{ReLU}(\mathsf{v}_j)$. The division is performed using a garbled circuit.

Tab.11 gives our benchmarks for NN Training. Over SecureML [75], we have improvements in the range 2.7×-3.46× for LAN and 2.4×-2.8× for WAN. Here the improvement is further boosted with our implementation of the softmax function that requires 2 online rounds as opposed to 4 rounds in SecureML.

Parameter	Ref.	LAN		WAN	
		n = 100	n = 900	n = 100	n = 900
Runtime (ms)	[75]	8.68	8.77	1,759.92	1,759.95
	ABY2.0	**2.66**	**2.66**	**744.12**	**744.12**
TP (queries/min)	[75]	62.02	40.89	0.19	0.12
	ABY2.0	**30,796.99**	**30,795.17**	**92.39**	**91.57**

Table 12: Comparison of the online runtime and throughput of ABY2.0 and SecureML [75] for NN Inference. Best results in bold. n is the number of features.

Tab. 12 gives our benchmarks for NN Inference. Here we improve the online runtime of SecureML [75] by a factor of 3.3× in LAN and 2.4× in WAN. Regarding the online throughput, we observe huge improvements in the range 496×–754× for both LAN and WAN. This improvement is primarily due to our efficient dot product protocol from §5.1 which has a dimension-independent online communication.

Setup Costs for PPML. We incur a minimal overhead of just 1.6% over SecureML [75] in terms of communication in the setup phase for Logistic Regression, while the overhead is 0.7% for the case of Neural Networks. The overhead results from the expensive communication required by our activation functions (Sigmoid and ReLU) over the garbled circuit-based solutions of SecureML [75].

Acknowledgements

Arpita Patra would like to acknowledge financial support from SERB MATRICS (Theoretical Sciences) Grant 2020 and Google India AI/ML Research Award 2020. Ajith Suresh would like to acknowledge financial support from Google PhD Fellowship 2019.

This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by DFG — SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and by BMBF and HMWK within ATHENE.

References

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys*, 2018.

[2] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, 2015.

[3] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In *Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019.

[4] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *IEEE S&P*, 2017.

[5] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*, 2013.

[6] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, 2007.

[7] A. Beaumont-Smith and C. Lim. Parallel prefix adder design. In *IEEE Symposium on Computer Arithmetic*, 2001.

[8] D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[9] D. Beaver. Precomputing oblivious transfer. In *CRYPTO*, 1995.

[10] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, 1990.

[11] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P*, 2013.

[12] A. Ben-Efraim, M. Nielsen, and E. Omri. Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing. In *ACNS*, 2019.

[13] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, 1988.

[14] M. Blanton and F. Bayatbabolghani. Efficient server-aided secure two-party function evaluation with applications to genomic computation. In *PETS*, 2016.

[15] M. Blanton and P. Gasti. Secure and efficient protocols for iris and fingerprint identification. In *ESORICS*, 2011.

[16] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, C. Wierzynski, and H. Yalame. MP2ML: A mixed-protocol machine learning framework for private inference. In *ARES*, 2020.

[17] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.

[18] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 2012.

[19] J. Boyar, P. Matthews, and R. Peralta. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 2013.

[20] J. Boyar and R. Peralta. Concrete multiplicative complexity of symmetric functions. In *International Symposium on Mathematical Foundations of Computer Science*, 2006.

[21] J. Boyar and R. Peralta. A small depth-16 circuit for the AES S-box. In *IFIP International Information Security Conference*, 2012.

[22] L. T. Brandão, N. Christin, and G. Danezis. Toward mending two nation-scale brokered identification systems. In *PETS*, 2015.

[23] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. Privacy-preserving remote diagnostics. In *CCS*, 2007.

[24] J. Bringer, H. Chabanne, M. Favre, A. Patey, T. Schneider, and M. Zohner. GSHADE: Faster privacy-preserving distance computation and biometric identification. In *IH&MMSEC*, 2014.

[25] P. Bunn and R. Ostrovsky. Secure two-party k-means clustering. In *CCS*, 2007.

[26] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *CCS*, 2018.

[27] M. Byali, H. Chaudhari, A. Patra, and A. Suresh. Flash: Fast and robust framework for privacy-preserving machine learning. In *PETS*, 2020.

[28] M. Byali, A. Joseph, A. Patra, and D. Ravi. Fast secure computation for small population over the Internet. In *CCS*, 2018.

[29] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. *Journal of Computer Security*, 2016.

[30] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: High throughput 3PC over rings with application to secure prediction. In *CCSW*, 2019.

[31] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *NDSS*, 2020.

[32] J. I. Choi, D. J. Tian, G. Hernandez, C. Patton, B. Mood, T. Shrimpton, K. R. Butler, and P. Traynor. A hybrid approach to secure function evaluation using SGX. In *ASIACCS*, 2019.

[33] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. Spd$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO*, 2018.

[34] R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz. Efficient multi-party computation over rings. In *EUROCRYPT*, 2003.

[35] I. Damgård, D. Escudero, T. K. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE S&P*, 2019.

[36] I. Damgård, M. Geisler, and M. Krøigaard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 2008.

[37] I. Damgård, C. Orlandi, and M. Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In *CRYPTO*, 2018.

[38] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[39] D. Demmler, T. Schneider, and M. Zohner. ABY – A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[40] G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner. Pushing the communication barrier in secure computation using lookup tables. In *NDSS*, 2017.

[41] ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils.

[42] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *PETS*, 2009.

[43] D. Evans, Y. Huang, J. Katz, and L. Malka. Efficient privacy-preserving biometric identification. In *NDSS*, 2011.

[44] O. Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[45] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[46] S. D. Gordon, S. Ranellucci, and X. Wang. Secure computation with low communication from cross-checking. In *ASIACRYPT*, 2018.

[47] D. M. Harris. A taxonomy of parallel prefix networks. In *Asilomar Conference on Signals, Systems and Computers*, 2003.

[48] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *TCC*, 2008.

[49] W. Henecka, S. Kögl, A. R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for automating secure two-party computations. In *CCS*, 2010.

[50] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[51] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[52] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint*, 2018. http://arxiv.org/abs/1803.05961.

[53] N. Husted, S. Myers, A. Shelat, and P. Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *CCS*, 2013.

[54] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In *STOC*, 1989.

[55] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, 2003.

[56] M. H. S. Javadi, M. H. Yalame, and H. R. Mahdiani. Small constant mean-error imprecise adder/multiplier for efficient VLSI implementation of mac-based applications. *IEEE Transactions on Computers*, 2020.

[57] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security*, 2018.

[58] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert. Mobile private contact discovery at scale. In *USENIX Security*, 2019.

[59] M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, 2016.

[60] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, 2018.

[61] M. Keller, P. Scholl, and N. P. Smart. An architecture for practical actively secure MPC with dishonest majority. In *CCS*, 2013.

[62] J. Kilian. Founding cryptography on oblivious transfer. In *STOC*, 1988.

[63] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO*, 2013.

[64] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*, 2009.

[65] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 2013.

[66] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.

[67] N. Koti, M. Pancholi, A. Patra, and A. Suresh. SWIFT: super-fast and robust privacy-preserving machine learning. *IACR Cryptology ePrint Archive*, 2020. https://eprint.iacr.org/2020/592.

[68] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, 2012.

[69] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 1980.

[70] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. http://yann.lecun.com/exdb/mnist/.

[71] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, 2007.

[72] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 2009.

[73] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In *CCS*, 2018.

[74] P. Mohassel, M. Rosulek, and Y. Zhang. Fast and secure three-party computation: Garbled circuit approach. In *CCS*, 2015.

[75] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, 2017.

[76] M. Naor and B. Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 2005.

[77] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*, 1999.

[78] S. Ohata and K. Nuida. Communication-efficient (client-aided) secure two-party protocols and its application. In *FC*, 2020.

[79] E. Orsini, N. P. Smart, and F. Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In *CT-RSA*, 2020.

[80] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[81] A. Patra and D. Ravi. On the exact round complexity of secure three-party computation. In *CRYPTO*, 2018.

[82] A. Patra, P. Sarkar, and A. Suresh. Fast actively secure OT extension for short secrets. In *NDSS*, 2017.

[83] A. Patra, T. Schneider, A. Suresh, and H. Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. *IACR Cryptology ePrint Archive*, 2020. https://eprint.iacr.org/2020/1225.

[84] A. Patra and A. Suresh. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *NDSS*, 2020.

[85] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security*, 2015.

[86] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, 2009.

[87] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai. Efficient circuit-based PSI with linear communication. In *EUROCRYPT*, 2019.

[88] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient circuit-based PSI via cuckoo hashing. In *EUROCRYPT*, 2018.

[89] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security*, 2018.

[90] D. Rathee, T. Schneider, and K. Shukla. Improved multiplication triple generation over rings via RLWE-based AHE. In *CANS*, 2019.

[91] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *ASIACCS*, 2018.

[92] D. Rotaru and T. Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *INDOCRYPT*, 2019.

[93] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *ICISC*, 2009.

[94] S. Sharma, C. Xing, and Y. Liu. Privacy-preserving deep learning with SPDZ. In *The AAAI Workshop on Privacy-Preserving Artificial Intelligence*, 2019.

[95] Stanford. CS231n: Convolutional neural networks for visual recognition. https://cs231n.github.io/convolutional-networks/.

[96] O. Tkachenko, C. Weinert, T. Schneider, and K. Hamacher. Large-scale privacy-preserving statistical computations for distributed genome-wide association studies. In *ASIACCS*, 2018.

[97] M. H. Yalame, M. H. Farzam, and S. B. Sarmadi. Secure two-party computation using an efficient garbled circuit by reducing data transfer. In *Applications and Techniques in Information Security (ATIS)*, 2017.

[98] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[99] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, 2015.

[100] Y. Zhang, A. Steele, and M. Blanton. PICCO: A general-purpose compiler for private distributed computation. In *CCS*, 2013.

[101] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

A Preliminaries

A.1 Oblivious Transfer (OT)

In a 1-out-of-n Oblivious Transfer [54, 76] (OT) over ℓ-bit messages, the sender S inputs n messages $(x_1,\ldots,x_n)$ each of length ℓ bits, while the receiver R inputs the choice $c \in \{1,\ldots,n\}$. R receives x_c as output while S receives $\perp$ as output. The privacy guarantee is that S learns nothing about c, while R learns nothing about the inputs of S other than x_c. We use $\mathsf{n\text{-}OT}_\ell^m$ to denote m instances of 1-out-of-n OT on ℓ bit inputs.

OT is a fundamental building block for MPC [62] and requires expensive public-key cryptography [54]. The technique of OT Extension [5, 55, 63, 82] allows us to generate many OTs from a small number (equal to the security parameter) of base OTs at the expense of symmetric-key operations alone. This reduces the cost of OT mainly to highly efficient symmetric-key primitives. Concretely, the OT Extension implementation of [5] generates around 1 million $\mathsf{2\text{-}OT}_\ell^1$ per second with passive security. An orthogonal line of work considered pre-computation of OT [9], where all the cryptographic operations can be shifted to a setup phase, independent of the function to be evaluated. This technique enables a very efficient online phase for protocols that use OT. In the semi-honest setting, the state-of-the-art solution for OT extension [5] has communication $\kappa+2\ell$ bits per OT for $\mathsf{2\text{-}OT}_\ell^1$ where κ denotes the computational security parameter.

A correlated OT (cOT) [5] is a variant of the traditional OT where the sender's input messages are correlated. In a cOT, the sender inputs a correlation function $f()$ and obtains the message pair $(x_0 \in_R \{0,1\}^\ell, x_1 = f(x_0))$ as the output. The receiver, on the other hand, inputs her choice c and obtains x_c as output. We use cOT_ℓ^m to denote m instances of 1-out-of-2 correlated OT on ℓ bit inputs. In the semi-honest setting, cOT_ℓ^1 has communication $\kappa+\ell$ bits [5].

A.2 Secure 2PC

Homomorphic Encryption (HE). The homomorphic property allows us to compute a ciphertext from a set of ciphertexts such that the plaintext underlying the former is a function of the underlying plaintexts of the latter. Towards this, one party called client generates a key-pair $(\mathsf{pk},\mathsf{sk})$ for the HE scheme and sends pk to the other party called server. To perform a secure computation operation, the client encrypts its data using pk and sends this to the server. Now the server can locally compute the ciphertext corresponding to the operation and return the encrypted result to the client. The client can now decrypt the received ciphertext using her private key sk. An *additively HE* allows us to generate the ciphertext corresponding to the sum of the underlying plaintexts by doing operations on the ciphertexts. Prominent examples of additively HE schemes are Paillier [80], DGK [36] and RLWE-AHE [90]. On the other hand, fully homomorphic encryption schemes allow arbitrary computations under encryption but are less efficient. See [1] for a more detailed description.

Garbled Circuits (GC). In the two-party setting, Yao's garbled circuit protocol [72, 98] provides a constant-round solution. This method is particularly useful in high-latency networks like the Internet. Here, one party called *garbler*

generates the garbled circuit (GC) corresponding to the function to be evaluated. On a high level, garbling the circuit consists of associating two keys per wire corresponding to the bit values of $\{0,1\}$ and preparing garbled tables corresponding to each gate in the circuit. The garbler then sends the GC to the other party called *evaluator*. The evaluator, upon obliviously obtaining the keys corresponding to the inputs via OT, evaluates the GC and obtains the output.

Today's most efficient solution for garbled circuits is the combination of point-and-permute [10], free-XOR [66], fixed-key AES [11], and half-gates [99]. With these optimizations, each AND gate requires communication 2κ bits in the setup phase, and XOR gates have no communication. GC-based protocols perform in the online phase symmetric-key operations for each AND gate and need substantial memory to store the garbled tables. To avoid storing the garbled tables, their generation and transfer can be pipelined [49,51], but this shifts all the setup communication to the online phase.

Secret Sharing (SS). In the SS-based protocols, two parties compute a function in a secret-shared manner. Here, for every wire with value v, party P_i for $i \in \{0,1\}$ holds an additive sharing of the value denoted by $[\mathsf{v}]_i$ such that $\mathsf{v} = [\mathsf{v}]_0 + [\mathsf{v}]_1 \pmod{2^\ell}$. All the linear gates can be evaluated non-interactively. To securely evaluate a multiplication gate, parties use Beaver's [8] circuit randomization technique where the additive sharing of a random arithmetic triple is generated in the setup phase (cf. §3.1.1). The shares of the triple are then used in the online phase to compute the shares of the product. This requires communication of 4 ring elements per multiplication gate in the online phase. Later, [12] reduced online communication to 2 ring elements using a function-dependent preprocessing.

In this line of work, the GMW protocol [45] takes a function represented as Boolean circuit (i.e., $\ell = 1$) and the values are secret-shared using XOR-based secret sharing. To pre-compute a multiplication triple $(\mathsf{c}_1 \oplus \mathsf{c}_2) = (\mathsf{a}_1 \oplus \mathsf{a}_2) \wedge (\mathsf{b}_1 \oplus \mathsf{b}_2)$, the solution of [5] which uses 1-out-of-2 OT, requires 2κ bits of communication. As shown in [40], this cost can be improved by factor $1.2\times$ by using the 1-out-of-N OT extension of [63].

A.3 Comparison with Turbospeedz [12] and [78]

Comparison with Turbospeedz [12]. For the 2-input multiplication, Turbospeedz [12] presented a protocol that reduces the online communication of SPDZ-style protocols from 4 to 2 ring elements using a function-dependent preprocessing. Turbospeedz first executes a SPDZ-like preprocessing where random multiplication triples are generated. These triples are then associated to the multiplication gates using additional values that they call "external values" (cf. [12], §3.2). On the contrary, we obtain the preprocessing data directly and hence save communication of 4 ring elements as well as storage of 5 ring elements when compared with Turbospeedz. Tab. 13 provides the communication and storage required for the 2-input multiplication protocol of ABY [39], Turbospeedz [12] and ABY2.0.

Phase	Parameter	ABY [39]	Turbospeedz [12]	**ABY2.0**
Setup	Storage	3ℓ	9ℓ	4ℓ
	Communication	$\lvert\mathsf{Triple}\rvert$	$\lvert\mathsf{Triple}\rvert + 4\ell$	$\lvert\mathsf{Triple}\rvert$
Online	Storage	5ℓ	5ℓ	$\mathbf{3\ell}$
	Communication	4ℓ	$\mathbf{2\ell}$	$\mathbf{2\ell}$
Total	Storage	8ℓ	14ℓ	7ℓ
	Communication	$\lvert\mathsf{Triple}\rvert + 4\ell$	$\lvert\mathsf{Triple}\rvert + 6\ell$	$\lvert\mathsf{Triple}\rvert + 2\ell$

Table 13: Comparison of ABY2.0 with ABY [39] and Turbospeedz [12] in terms of storage and communication for a single multiplication. All values are given in bits. $\lvert\mathsf{Triple}\rvert$ denotes the communication required to generate a multiplication triple. Best values for the online phase are marked in bold.

For the multi-input multiplication (fan-in of N), the tree-based method (multiplying N elements by taking two at a time) requires $\log_2(N)$ rounds for both ABY [39] and Turbospeedz [12], while it requires communication of $4(N-1)$ ring elements for ABY [39] and $2(N-1)$ elements for Turbospeedz [12] in the online phase.

Comparison with [78]. Recently, [78] proposed round-efficient solutions for multi-input multiplication using a preprocessing for which the communication cost grows exponentially with the fan-in of the multiplication gate. However, for an N-input multiplication, [78] requires an online communication of $2N-2$ ring elements. On the contrary, ABY2.0 requires only an online communication of 2 ring elements and the preprocessing cost remains same as that of [78]. Note that since the preprocessing cost grows exponentially with the number of inputs to the multiplication gate, [78] considered only up to 5-input multiplication gates in their work. In our work, we use three and four input multiplication gates.

MULT *when input parties are the computing parties:* For the case of a two-input multiplication gate, [78] considered a special case where the input parties are the computing parties (cf. [78], §3.4). For this case, [78] proposed a protocol for which the online communication is 2 ring elements. For the same setting, we observe that our solution results in a protocol with zero online communication. To see this, recall the online phase of our multiplication protocol $\mathsf{MULT}(\langle\mathsf{a}\rangle, \langle\mathsf{b}\rangle)$ (Fig. 2). The modified protocol is as follows: During the online phase, party P_i for $i \in \{0,1\}$ locally computes $[\mathsf{ab}]_i = i \cdot \Delta_{\mathsf{ab}} - \Delta_{\mathsf{a}}[\delta_{\mathsf{b}}]_i - \Delta_{\mathsf{b}}[\delta_{\mathsf{a}}]_i + [\delta_{\mathsf{ab}}]_i$. Now to generate $\langle\cdot\rangle$-shares corresponding to $\mathsf{y} = \mathsf{ab}$, the parties locally set $[\delta_{\mathsf{y}}]_i = -[\mathsf{ab}]_i$ and $\Delta_{\mathsf{y}} = 0$. It is easy to see that $\mathsf{y} = \Delta_{\mathsf{y}} - [\delta_{\mathsf{y}}]_0 - [\delta_{\mathsf{c}}]_1 = 0 - ([\mathsf{ab}]_0 + [\mathsf{ab}]_1) = \mathsf{ab}$.