Scalable Multi-Party Computation Protocols for Machine Learning in the Honest-Majority Setting

Fengrun Liu
University of Science and Technology of China & Shanghai Qi Zhi Institute

Xiang Xie
Shanghai Qi Zhi Institute & PADO Labs

Yu Yu
Shanghai Jiao Tong University & State Key Laboratory of Cryptology, P. O. Box 5159, Beijing, 100878, China

Abstract

In this paper, we present a novel and scalable multi-party computation (MPC) protocol tailored for privacy-preserving machine learning (PPML) with semi-honest security in the honest-majority setting. Our protocol utilizes the Damgård-Nielsen (Crypto'07) protocol with Mersenne prime fields. By leveraging the special properties of Mersenne primes, we are able to design highly efficient protocols for securely computing operations such as truncation and comparison. Additionally, we extend the two-layer multiplication protocol in ATLAS (Crypto'21) to further reduce the round complexity of operations commonly used in neural networks.

Our protocol is very scalable in terms of the number of parties involved. For instance, our protocol completes the online oblivious inference of a 4-layer convolutional neural network with 63 parties in 0.1 seconds and 4.6 seconds in the LAN and WAN settings, respectively. To the best of our knowledge, this is the first fully implemented protocol in the field of PPML that can successfully run with such a large number of parties. Notably, even in the three-party case, the online phase of our protocol is more than $1.4\times$ faster than the Falcon (PETS'21) protocol.

1 Introduction

In recent years, machine learning (ML) has gained immense popularity across various domains, including medicine, banking, recommendation systems, and biometric authentication. However, the widespread use of ML models has raised significant privacy concerns. Moreover, many companies offer pre-trained ML models, and performing inference operations on sensitive data introduces additional privacy considerations. Regulations like HIPAA and GDPR impose significant constraints on institutions aggregating personal data from individuals. To enable effective machine learning while preserving privacy, the ideal approach is for individuals to retain their personal information locally while collaborating to train models in a secure multi-party manner.

Secure Multi-Party Computation (MPC) [3, 21, 47] enables a group of n parties to collectively compute a function on their private inputs while preserving the privacy of the inputs. MPC offers a promising approach to address these privacy challenges in ML. It enables privacy-preserving machine learning (PPML) by performing secure computations on distributed datasets without exposing the individual data points or the model itself. This allows multiple parties to collaborate on training or inference tasks while keeping their data private. By leveraging cryptographic techniques and secure protocols, MPC-based PPML ensures that sensitive information remains protected throughout the computation process.

In the context of PPML, there have been significant research efforts to enhance secure multi-party computation protocols in different settings. In the dishonest-majority setting, the majority of PPML protocols concentrate on the two-party scenario [7, 12, 24, 31, 33, 35, 38, 40, 41, 42]. On the other hand, in the honest-majority setting, existing PPML protocols only consider the three-party [8, 12, 29, 34, 39, 44, 45] and four-party cases [4, 9, 13, 28], where the protocols tolerate one corrupted party. However, many applications like federated learning involve large numbers of participants, yet existing privacy-preserving machine learning protocols focus on 2-4 parties only, falling short of real-world needs. There is a pressing need to develop scalable protocols that can accommodate machine learning with many distributed data owners.

In theory, there are general-purpose MPC protocols, such as those mentioned in [10, 17, 18, 26], that can support an arbitrary number of parties. However, these protocols lack efficient methods for securely computing non-linear functions like truncation and comparison. This limitation is the primary reason why they do not report specific performance results when applied to a large number of parties in PPML.

Previous protocols often utilize power-of-two rings $\mathbb{Z}_{2^\ell}$ and prime fields $\mathbb{F}_p$. Power-of-two rings, specifically those with $\ell = 32$ or 64, are particularly favored due to their superior computation efficiency. This efficiency stems from the compatibility of the ring operations with native datatypes. Prime fields offer significant advantages when it comes to leveraging the fast protocol proposed by Damgård and Nielsen [17], hereafter known as the DN protocol, which is based on Shamir Secret Sharing.

In PPML, the secure computation of truncation and comparison functions plays a vital role. Practical truncation protocols are based on the approaches (and their extensions) proposed in [35] and [6] in the power-of-two rings and prime fields setting, respectively. However, these methods result in a significant gap between the shares and the secret, necessitating the use of either a large modulus or low precision to mitigate truncation errors. For the secure computation of the comparison function, protocols (such as [15, 34]) typically rely on the computation of boolean circuits with bit shares. These methods also introduce a large gap between the shares and secrets. Moreover, the online round complexity can be as high as $O(\log \ell)$. Note that [12, 20] show how to reduce the gap of the truncation protocol in the power-of-two rings setting, but the round complexity of their comparison protocols is still high. Additionally, [32] shows how to mitigate the gap in the comparison protocol in the prime fields setting. But the gap in the truncation protocol still persists.

1.1 Our Contribution

We design and implement a general framework of scalable multi-party computation protocols for privacy-preserving inference, where the model is secret-shared among all data owners, with semi-honest security in the honest majority setting. This establishes a foundation for realizing practical large-scale privacy-preserving collaborative training in the future. Our schemes leverage the DN [17] protocol with Mersenne prime fields. I.e., the prime is in the form $p = 2^\ell - 1$. It is worth noting that we are the first to take advantage of the unique properties of Mersenne primes to optimize truncation and comparison from a theoretical standpoint. The key contributions of this paper are as follows.

1. We propose a novel protocol to securely compute the truncation function in the setting of prime fields with Mersenne primes. This new protocol reduces the gap between shares and the secret to just 1 bit. As a result, It allows us to use a smaller modulus or achieve higher precision, thereby directly improving both computation and communication efficiency. We further combine it with the DN multiplication protocol, resulting in an efficient protocol for fixed-point multiplication with 1 round online complexity.

2. We propose an efficient protocol to securely compute the bitwise comparison function in the setting of Mersenne prime fields. It eliminates the gap between shares and the secret with 1 round online complexity. To achieve this, besides utilizing the special properties of Mersenne primes, we also present an optimized Prefix-`OR` protocol with only 1 round online complexity. This new Prefix-`OR` protocol works for any prime field, which is of independent interest.

3. We simplify and extend the techniques of evaluating two-layer multiplication in [22], to securely compute ReLU (Maxpool) operations in neural networks, resulting in efficient protocols with 3 rounds ($3\log m$ rounds) online complexity.

4. We implement our framework with C++ and conduct experiments of oblivious inference for neural networks with varying numbers of parties, ranging from 3 to 63. For example, when performing inference on a 4-layer convolutional neural network with 63 parties, the online (preprocessing) time is 0.4 s (2.4 s) and 4.3 s (12.3 s) in the LAN and WAN setting, respectively. To the best of our knowledge, this is the first fully implemented protocol in the field of PPML that can successfully run with such a large number of parties. Even in the three-party case, our protocol outperforms the Falcon protocol [45] in terms of online phase speed, being more than $1.4\times$ faster.

An overview of the different protocols considered in this work is provided in Appendix A

1.2 Overview of Techniques

Let p be a Mersenne prime, which is in the form of $p = 2^\ell - 1$ for some prime ℓ. It is well known that the distribution of Mersenne primes is quite sparse. However, for practical purposes in privacy-preserving machine learning, it is fortunate that values of ℓ such as 31, 61, and 127 are often sufficient. These specific values of ℓ are typically suitable for most use cases in PPML, allowing for efficient computations within the domain.

Advantages of Mersenne Primes. A key observation in this paper is that Mersenne primes are very close to power-of-two numbers. Mersenne primes exhibit similar properties to power-of-two numbers, which leads to several advantages in computations. In particular, modular operations performed over Mersenne primes involve simple operations such as bit shifts and additions, making them highly efficient. When encoding a signed integer $a \in [-(p-1)/2, (p-1)/2]$ into a representation $\bar{a}$ within the range of $[0, p-1]$ in $\mathbb{F}_p$, the most significant bit (MSB) of $\bar{a}$ effectively indicates the sign bit of a. This is due to the fact that $(p-1)/2 = 2^{\ell-1} - 1$. The plain truncation protocol in $\mathbb{F}_p$ is quite simple when p is a Mersenne prime. Truncating $\bar{a} \in \mathbb{F}_p$ by d bits is just shifting the bits of $\bar{a}$ down by d positions and filling the top d bits with the MSB of $\bar{a}$.

Probabilistic Truncation with Only 1-Bit Gap. Let $\mathrm{Trunc}_d(x)$ denote the truncation of $x \in \mathbb{F}_p$ with d bits. The starting point is the method presented in ABY3 [34]. In a nutshell, they utilize the property that $\mathrm{Trunc}_d(z) =$

$\text{Trunc}_d(x = z + r) - \text{Trunc}_d(r) + e$ for small z with high probability for uniformly random $r \in \mathbb{F}_p$, where $|e| \leq 1$ is some rounding error. The constraints on z and the probability mentioned in the previous statement are imposed due to certain cases where the equation does not hold. This deviation from the equation occurs when there is a "wapping around" event.

The "wapping around" event happens in two cases. 1) z, r represent two postive integers, but $x = z + r$ represents a negative integer. 2) z, r represent two negative integers, but $x = z + r$ represents a positive integer. This is due to the implicit modulo operation involved in operations within the prime field $\mathbb{F}_p$. The constraints on z and r make this bad case occur only with a small probability. In conclusion, for $z \in [0, 2^{\ell_z}] \cup [p - 2^{\ell_z}, p)$, the equation holds with probability $1 - \frac{2^{\ell_z+1}}{p}$ for some $\ell_z > 0$.

It is easy to see that the gap between z and p should be large enough to make the equation hold. In this paper, we fully leverage the properties of Mersenne primes to narrow the gap between z and p to just 1 bit.

We consider $z \in [0, (p-1)/2]$ representing a positive integer. Under this condition, only one specific case will trigger the "wrapping around" event. I.e., r represents a positive integer, but x represents a negative integer. In other words, $r_{\text{msb}} = 0 \wedge x_{\text{msb}} = 1$. In this case, let us look into the result of $\text{Trunc}_d(x = z + r)$. As described before, $\text{Trunc}_d(x = z + r)$ shifts down x by d bits and fills the first d bits with the MSB of x, which is bit 1. Surprisingly, the deviation in the equation is caused by the "misfilled" first d bits, and we simply remove this term to make the equation hold again in this case!

Removing the first d bits with all 1's is equivalent to substract $(2^d - 1) \cdot 2^{\ell-d} = (2^\ell - 2^{\ell-d}) = p - (2^{\ell-d} - 1)$. This is equivalent to adding $2^{\ell-d} - 1$ in the field. In the MPC setting, we remain to compute the share of $(1 - r_{\text{msb}}) \cdot x_{\text{msb}}$ to obliviously indicate whether the "wrapping around" event occurs or not. Fortunately, $x = z + r$ will be revealed in the protocol because r is uniformly random, and x_{msb} can be computed locally. Further, in the preprocessing phase, we will prepare the share of r_{msb}. Therefore the share of $(1 - r_{\text{msb}}) \cdot x_{\text{msb}}$ can be computed locally!

Note that our truncation protocol can be simply modified to support elements that represent negative integers as well. Let $\bar{z} \in [0, 2^{\ell-2}) \cap [p - 2^{\ell-2}, p)$, set $z = \bar{z} + 2^{\ell-2} \in [0, (p-1)/2]$. Then, we can easily obtain $\text{Trunc}_d(\bar{z})$ by applying our protocol on z. This is because $\text{Trunc}_d(z) \approx \text{Trunc}_d(\bar{z}) + 2^{\ell-d-2}$.

In PPML, the secure truncation protocol typically follows a secure multiplication protocol. Specifically, there is a need to securely compute the truncation of the product of two values, denoted as $\text{Trunc}_d(a \cdot b)$. This operation is often referred to as a secure fixed-point multiplication protocol. In the DN [17] multiplication protocol, $ab + r$ will be revealed anyway. This enables us to further reduce the online round complexity of secure fixed-point multiplication protocol to 1 round.

Round-Efficient Comparison. Our secure bitwise comparison protocol follows from [15]. The core and bottleneck of this protocol are to compute all the Prefix-`OR`s of a bit vector. Specifically, given a bit vector $(a_1, \ldots, a_\ell)$, compute $b_i = \bigvee_{j=1}^{i} a_j$ for $1 \leq j \leq \ell$. Several works [15, 36] of secure Prefix-`OR` protocols are proposed to optimize the online round complexity. These protocols have achieved constant online round complexity, but the actual number of rounds involved can still be relatively large. [6] proposed a small constant-round version that requires a large gap.

A key observation of this paper is that instead of computing all Prefix-`OR`s, we compute its dual problem Prefix-`AND`s. This is because computing $b_i = \bigvee_{j=1}^{i} a_j$ is identical to compute $\overline{b}_i = \bigwedge_{j=1}^{i} \overline{a}_j$, where $\overline{b}_i = 1 - b_i$ and $\overline{a}_i = 1 - a_i$. The Prefix-`AND`s problem is indeed to compute all prefix products $\overline{b}_i = \prod_{j=1}^{i} \overline{a}_j$ for $1 \leq j \leq \ell$, which could be securely computed with 1 round online complexity using the unbounded fan-in multiplication protocol from [1, 6, 15, 36].

With this efficient secure comparison protocol, we could design a fast secure DReLU protocol. DReLU is the derivative of the non-linear activation ReLU function. It serves as a core building block in PPML. Given a number a, $\text{DReLU}(a) = 0$ if $a < 0$, and $\text{DReLU}(a) = 1$ otherwise. In other words, $\text{DReLU}(a) = 1 - a_{\text{msb}}$. Follow the idea from SecureNN [44], we have $a_{\text{msb}} = t_{\text{lsb}}$, where $t = 2a \in \mathbb{F}_p$, because p is odd.

In the preprocessing phase, we generate the shares of a random value r, denoted as $[r]_p$, and the shares of the bits of r, denoted as $[r]_{\mathrm{B}}$. In the online phase, $[y]_p = [t + r]_p$ is revealed, and one can compute $[t_{\text{lsb}}]_p = y_{\text{lsb}} \oplus [r_{\text{lsb}}]_p \oplus [(t + r \geq p)]_p$. Note that $t + r \geq p$ if and only if $y < r$. Since we already have $[r]_{\mathrm{B}}$, using the secure bitwise comparison protocol on bits results in an efficient secure DReLU protocol.

Vertorized Secure Two-Layer Multiplication. Many operations in neural networks can be derived from DReLU. For example, the ReLU functions can be represented as $\text{ReLU}(a) = \text{DReLU}(a) \cdot a$, and $\text{Max}(a, b) = \text{DReLU}(a - b) \cdot (a - b) + b$. We observe that the last step of secure DReLU involves securely XORing two shares of bits, which is essentially a secure multiplication. Therefore the ReLU and Max functions can be seen as two-layer multiplications. We adopt a vectorized version of the optimized two-layer multiplication protocol proposed in the recent work [22]. This approach allows us to further reduce the online round complexity associated with these operations.

1.3 Other Related Work

Truncation. The protocol of truncation over fields in the recent work from Escudero et al. [20] indeed originates from the previous works presented by Catrina and de Hoogh [6], which entails a large gap between the secrets and the actual modular in the field to assure the correctness with statistical security.

Authors in ABY3 [34] and [14] propose alternative methods for truncation over rings, but unfortunately their methods require a large gap or involve complex bitwise subprotocols in logarithmic rounds. The current state-of-the-art for truncation in the ring case [12, 20] uses the fact that the input is positive to reduce the gap to 1-bit without any bitwise subprotocols. Our method for truncation over fields is similar to this idea yet with a more intuitive and simple way of detecting and correcting the error caused by the "wrap around" event. Moreover, we seamlessly combine the truncation and DN multiplication to instantiate the fixed-point multiplication.

Comparison. Catrina and de Hoogh [6] proposed a (small) constant-round comparison protocol that requires a large gap for correctness. The most recent work for comparison is Rabbit [32], which removes the need for a large gap but depends on several logarithmic bitwise subprotocols. The comparison with no gap requirement was first studied in [15], and optimized by [36], with non-negligible constant cost, in which the main bottleneck is the bitwise less-than protocol involved a very costly circuit for evaluating prefix-OR. In our work, we propose a naive approach to compute the prefix-OR in the field setting, directly optimizing the bitwise less-than protocol in [15]. Various protocols for complex computations, e.g. deterministic truncation and DReLU, can benefit from this efficient bitwise less-than primitive.

1.4 Paper Organization

We organize the paper as follows: In Section 2, we provide an overview of the known techniques used in Damgård-Nielsen protocols [19]. In Section 3, we discuss our secure truncation protocol with only a 1-bit gap and propose a detailed protocol for fixed-point multiplication. Then, in Section 4, we introduce the optimized bitwise primitives that we have developed. Section 5, presents our efficient building blocks, including DReLU, ReLU, and Maxpool, which are essential for neural networks. Finally, in Section 6, we analyze the efficiency of our frameworks based on the results obtained from our implementation.

2 Preliminary

2.1 Model.

We use $P_1,\dots,P_n$ to denote n parties, respectively, which are to do the secure computation. We assume that each pair of parties share an authenticated channel and the parties have an authenticated broadcast channel, and the corrupted parties are assumed to be poly-time bounded. We describe our protocols in the so-called "honest-but-curious" model with passive security in the honest-majority setting, but standard techniques will be applicable to make our protocols robust with malicious security.

2.2 The Field over Mersenne Primes.

A Mersenne prime is a prime number that can be expressed in the form $M_\ell = 2^\ell - 1$, where ℓ is a prime number. Examples of Mersenne primes include $\ell = 31, 61, 127$. Throughout the rest of the paper, we will work with a fixed Mersenne prime $p = 2^\ell - 1$ to establish a finite field $\mathbb{F}_p$ on which most of our computations will be performed. The size of an element in $\mathbb{F}_p$ is denoted by ℓ. Let E be the set $\{-1,0,1\}$ and $2E$ be the set $\{0,\pm1,\pm2\}$ for brevity.

2.3 Secret Sharing.

This work is based on Shamir's (t,n)-threshold scheme [43] where n is the number of parties and t denotes the number of corrupted parties. For ease of simplicity, we let $n \geq 2t+1$. Each party is assigned a unique element $\alpha_i \in \mathbb{F}_p$ as the identity. For notational convenience, we index P_i by the identity $\alpha_i = i$.

By a t-polynomial we mean a polynomial $f(X) \in \mathbb{F}_p[X]$ of degree at most t. To share a secret $x \in \mathbb{F}_p$ with degree t, a uniformly random t-polynomial with $f(0) = x$ is chosen, and P_i is given the share $x_i = f(\alpha_i)$. We use $[x]_p$ to denote a degree-t sharing, such that $[x]_p = (x_1,\dots,x_n)$ where party P_i is holding x_i. The scheme is t-`privacy` and $(t+1)$-`reconstruction` such that any $\leq t$ shares jointly leak no information about the secret and any $\geq t+1$ shares can uniquely reconstruct the secret. Let $P_{\texttt{pking}}$ be the party that all parties agree on in the beginning. We write $x \leftarrow \Pi_{\text{Reveal}}([x]_p)$ during which each party sends its share to $P_{\texttt{pking}}$, and $P_{\texttt{pking}}$ reconstructs x and sends it back to other parties. This procedure is a dominant factor of the complexity, so we measure the round complexity of a protocol by the number of rounds of parallel invocations of Π_{Reveal}.[1] And the communication complexity is measured by the number of field elements sent by each party.

If $[a]_p$ and $[b]_p$ are degree-t sharings, multiplying directly yields a degree-$2t$ sharing. We use $\langle y \rangle_p$ to denote a degree-$2t$ sharing, which requires at least $2t+1$ shares for interpolation. In the honest-majority setting, the honest parties could recover y since $2t+1 < n$. Similarly, we write $y \leftarrow \Pi_{\text{Reveal}}(\langle y \rangle_p)$.

The Shamir's scheme allows to compute $[a+b \mod p]_p$ and $\langle a\cdot b \mod p\rangle_p$ from $[a]_p$ and $[b]_p$ with no communication. For simplicity, we write $[a+b]_p = [a]_p + [b]_p$ and $\langle ab\rangle_p = [a]_p \cdot [b]_p$.

2.4 Review: DN Multiplication.

Damgård and Nielsen proposed the most renowned DN protocol [17] in the honest majority setting for semi-honest security. The protocol consists of 4 phases: Preprocessing Phase, Input Phase, Online Phase, and Output Phase. Here we give a brief description of the Preprocessing Phase and the Online Phase.

[1]When measuring the round complexity in the preprocessing phase, we roughly measure a parallel unit broadcast as one round, which is common in the generation of randomness.

Offline Phase. In the offline phase, all parties need to prepare enough shared random numbers. Specifically, there are two kinds of random sharings in DN multiplication. The first kind is the degree-t sharing $[r]_p$. The second kind is a pair of random sharings $([r]_p, \langle r \rangle_p)$ of the same secret r, which is referred to as the random double sharings. We write the generation of random sharings as $[r]_p \leftarrow \Pi_{\text{Rand}}$ and $([r]_p, \langle r \rangle_p) \leftarrow \Pi_{\text{DoubleRand}}$.

In the original DN protocol, preparing a pair of double sharings requires the communication of $\frac{2n}{t+1} \approx 4$ field elements per party and $\frac{n}{t+1} \approx 2$ field elements per party for a random sharing $[r]_p$.

Online Phase. In the online phase, all parties need to evaluate addition and multiplication gates layer by layer. The addition gate can be locally computed. For a multiplication gate with input sharings $[x]_p$ and $[y]_p$, directly multiplying two degree-t sharings yields a degree-$2t$ sharing. After that, a pair of double sharings $([r]_p, \langle r \rangle_p)$ is consumed to securely reduce the degree. All parties execute the following steps.

1. All parties compute $\langle xy + r \rangle_p = [x]_p \cdot [y]_p + \langle r \rangle_p$.
2. $P_{\texttt{pking}}$ collects shares of $\langle xy + r \rangle_p$ and reconstructs the value $xy + r$. Then $P_{\texttt{pking}}$ sends it back to other parties.
3. All parties locally compute $[xy]_p = (xy + r) - [r]_p$.

We write the above DN multiplication protocol as $[xy]_p \leftarrow \Pi_{\text{Mult}}([x]_p, [y]_p)$. The online complexity of this protocol is 1 round with 2 field elements per party.

Reduce Communication Complexity with PRG. The communication complexity of distributing sharings can be reduced by utilizing pseudo-random generators (PRG). This trick has been used in previous works such as [22, 30, 37]. With PRG, the communication cost of distributing a degree-t sharing can be reduced by a factor of 2 and the cost of distributing a degree-$2t$ sharing is free. As a result, preparing a random sharing and a pair of double sharing both require the communication of 1 field element per party. And the improved DN multiplication protocol with PRG has communication complexity of 3 field elements per party.

2.5 Useful Techniques

Multiplication with Public Output by PRZS. DN multiplication protocol allows us to securely compute $[xy]_p$ from input $[x]_p$ and $[y]_p$ in one round. But using the pseudo-random sharing of zero (PRZS [11]) technique, we can securely compute the product of two sharings $[x]_p, [y]_p$ with public output xy in one round rather than invoking the original DN multiplication protocol that computes $[xy]_p$ and reveals the degree-t sharing. It works as follows: The parties generate a pseudo-random degree-$2t$ sharing $\langle 0 \rangle_p$ that can be done without interactions; each party computes $\langle z \rangle_p = [x]_p \cdot [y]_p + \langle 0 \rangle_p$; then reveals the degree-$2t$ sharing $\langle z \rangle_p$ to obtain xy. We write $xy \leftarrow \Pi_{\text{MultPub}}([x]_p, [y]_p)$ that can be done in 1 round with communication of 2 field elements per party (in the online phase).

Unbounded Fan-in Prefix-products. [1] proposed a technique to do unbounded fan-in multiplication in constant rounds. We follow the protocols in [6, 15, 36] to compute all prefix-products $b_i = \Pi_{j=1}^{i} a_j$ for $i = 1, \ldots, t$ as follows: In the preprocessing phase, parties prepare t correlated pairs of random sharings $([r_i]_p, [r_i']_p)$ where $r_1' = r_1^{-1}$ and $r_i' = r_{i-1} r_i^{-1}$ for $i > 1$ such that $r_i \Pi_{j=1}^{i} r_i' = 1$, with communication of $7t$ field elements per party. In the online phase:

1. For $i = 1, \ldots, t$: Compute $c_i \leftarrow \Pi_{\text{MultPub}}([a_i]_p, [r_i']_p)$.
2. For $i = 1, \ldots, t$: Compute locally $[b_i]_p = [r_i]_p \cdot \prod_{j=1}^{i} c_j$.

We write it as $[b_1]_p, \ldots, [b_t]_p \leftarrow \Pi_{\text{PreMult}}([a_1]_p, \ldots, [a_t]_p)$ which can be done in 1 round with $2t$ field elements per party. The generation of random values can be done in 2 rounds with $7t$ field elements per party ([36], [6]).

Random Bitwise-Sharings. [15] proposed a protocol to securely generate the shares of a uniformly random element along with the shares of its decomposed bits. Let r be an unknown uniform number such that $0 \le r = \sum_{i=0}^{t-1} 2^i r_i < p$ where each $r_i \in \{0, 1\}$, the parties generate each $[r_i]_p$ by computing $S(a) = \frac{\frac{a}{\sqrt{a^2}} + 1}{2}$ on a random sharing $[a]_p \leftarrow \Pi_{\text{Rand}}$, in which $\sqrt{a^2}$ is formulated to equal the unique element in $[1, (p-1)/2]$ so $a/\sqrt{a^2}$ is 1 or -1 in half chance when a is uniformly random in $\mathbb{F}_p^*$. Then the parties generate $[r]_p$ locally. We use $[r]_{\text{B}}$ to denote the generated $([r_0]_p, \ldots, [r_{t-1}]_p)$, and write the above process as $([r]_p, [r]_{\text{B}}) \leftarrow \Pi_{\text{SolvedBits}}$. Note that the original protocol in [15] invokes a bitwise less-than circuit and aborts if $r \ge p$ to guarantee the generated r lies in the field. Under the condition that it does not abort, r is uniformly random from $\mathbb{F}_p$ as desired. In this work, $p = 2^\ell - 1$ is a Mersenne prime, thus the probability that $r \ge p$ is $2^{-\ell}$. Hence, we omit the less-than circuit for larger primes, which can be done in 2 rounds with communication of $3t$ field elements per party in the preprocessing phase.

Beaver Triples. A Beaver triple [2] consists of three degree-t sharings $([a]_p, [b]_p, [c]_p)$ such that $c = a \cdot b$. Given two sharings $[x]_p, [y]_p$, one can compute $[xy]_p$ locally after revealing $u = x + a$ and $v = y + b$ as follows.

$$[xy]_p = u \cdot v - u \cdot [b]_p - v \cdot [a]_p + [c]_p$$

2.6 Security Model

We use $\mathcal{P} = \{P_1, \ldots, P_n\}$ to denote a set of n parties.We consider the honest-majority setting in the presence of static semi-honest adversaries. Such an adversary controls $t < n/2$ parties at the beginning of the protocol and follows the protocol specification exactly. However, it may try to learn more information than allowed by looking at the transcript of messages that the corrupted parties received and their internal state. We model and prove the security of our protocols under the universal composition (UC) framework [5]. Due to space constraints, we formally describe the functionalities mentioned in this work in Appendix B.

3 Our Fixed-Point Multiplication

3.1 Truncation in the Field

In this section, we consider signed integers in the interval $\left(-2^{\ell-1}, 2^{\ell-1}\right)$ where ℓ is the length of the Mersenne prime p. Given an interger $x \in \left(-2^{\ell-1}, 2^{\ell-1}\right)$, we can represent x with a corresponding integer in $\mathbb{F}_p$ by computing $(x \mod p) \in [0, p-1]$.

In Definition 3.1, we define the truncation operation (represented in the field) of integers. It is worth noting that the truncation operation is indeed a special division, in which the divisor is in the form 2^d. Intuitively, given some integer $d > 0$, let x be an integer with $|x| = x_1 \cdot 2^d + x_2$ and $0 \le x_2 < 2^d$, the truncation of x is x_1 if $x \ge 0$, and $p - x_1$ if $x < 0$.

Definition 3.1 (Truncation of Represented Integers)**.** *Given an element $x \in \mathbb{F}_p$, the truncation of x, denoted by* $\mathrm{Trunc}_d(x) \in \mathbb{F}_p$*, is defined as follows.*

$$\mathrm{Trunc}_d(x) = \begin{cases} \lfloor x/2^d \rfloor, & 0 \le x \le (p-1)/2 \\ p - \lfloor (p-x)/2^d \rfloor, & (p-1)/2 < x \le p-1 \end{cases}$$

Note that when $0 \le x \le (p-1)/2$, x represents a non-negative number, the truncation operation just removes the d least significant bits. x represents a negative number when $(p-1)/2 < x \le p-1$, and $p - x$ is the absolute value. The truncation operation first removes the d least significant bits of the absolute value and then represents the resulting negative integer in the field. It is easy to check that $\mathrm{Trunc}_d(x) = p - \mathrm{Trunc}_d(-x) = -\mathrm{Trunc}_d(-x) \in \mathbb{F}_p$.

In applications such as privacy-preserving machine learning, the truncation operation needs to be performed under MPC protocols. A common approach involves initially running a comparison circuit to determine the specific case, followed by executing the rounding operations. Running a comparison circuit with MPC is often a bottleneck in the process.

A key observation of this article is that when considering a Mersenne prime p in the form of $p = 2^\ell - 1$, the sign is entirely dependent on the most significant bit. This is because $(p-1)/2 = 2^{\ell-1} - 1$. x represents a negative integer if and only if the $(\ell-1)$-th bit is 1. Based on this, we present a more efficient truncation procedure as described in Theorem 3.1.

Intuitively, instead of running a comparison circuit with MPC, we simply shift the bits of x down by d positions and fill the top d bits with the MSB of x.

Theorem 3.1. *Let p be a Mersenne prime in the form of $p = 2^\ell - 1$. Let $x \in \mathbb{F}_p$ and $x_0, x_1, \dots x_{\ell-1}$ be the bit decomposition such that $x = \sum_{i=0}^{\ell-1} x_i \cdot 2^i$. We have:* $\mathrm{Trunc}_d(x) = \sum_{i=d}^{\ell-1} x_i \cdot 2^{i-d} + \sum_{i=\ell-d}^{\ell-1} x_{\ell-1} \cdot 2^i$.

Proof. If $0 \le x \le (p-1)/2$, then $x_{\ell-1} = 0$. We have $\mathrm{Trunc}_d(x) = \sum_{i=d}^{\ell-1} x_i \cdot 2^{i-d}$ as desired.

If $(p-1)/2 < x \le p-1$, due to the definition, we have

$$\mathrm{Trunc}_d(x) = 2^\ell - 1 - \lfloor (2^\ell - 1 - x)/2^d \rfloor$$

The $2^\ell - 1 - x$ term can be viewed as:

$$\begin{aligned} 2^\ell - 1 - x &= (2^\ell - 1 - \sum_{i=0}^{d-1} x_i \cdot 2^i - \sum_{i=d}^{\ell-1} x_i \cdot 2^i) \\ &= (2^\ell - \sum_{i=d}^{\ell-1} x_i \cdot 2^i - 2^d) + (2^d - 1 - \sum_{i=0}^{d-1} x_i \cdot 2^i) \end{aligned}$$

Note that $0 \le 2^d - 1 - \sum_{i=0}^{d-1} x_i \cdot 2^i \le 2^d - 1$, we have

$$\lfloor (2^\ell - 1 - x)/2^d \rfloor = 2^{\ell-d} - \sum_{i=d}^{\ell-1} x_i \cdot 2^{i-d} - 1.$$

Therefore, along with $x_{\ell-1} = 1$, we have

$$\begin{aligned} \mathrm{Trunc}_d(x) &= 2^\ell - 2^{\ell-d} + \sum_{i=d}^{\ell-1} x_i \cdot 2^{i-d} \\ &= \sum_{i=\ell-d}^{\ell-1} x_{\ell-1} \cdot 2^i + \sum_{i=d}^{\ell-1} x_i \cdot 2^{i-d} \end{aligned}$$

□

A Big Gap when Truncating. The SecureML paper [35] presents a truncation method in the 2PC setting. This technique, along with its extensions, is utilized by many other works [34, 44, 45] to perform secure truncation operations. For instance, the truncation technique in ABY3 [34], which can be extended to any number of parties, involves computing $\mathrm{Trunc}_d(z) \approx \mathrm{Trunc}_d(z = x + r) - \mathrm{Trunc}_d(r)$, where r is uniform. However, using this method, the above equation only holds for elements within the range of $\left[0, 2^{\ell_z}\right] \cup \left[p - 2^{\ell_z}, p\right)$ with probability $1 - \frac{2^{\ell_z+1}}{p}$ for some ℓ_z. Consequently, the size of the truncated value should be much smaller than p. In practice, we have to choose large p (say with 64-bit length) to avoid truncation errors of large magnitude. Essentially, we impose constraints on z to avoid a "wrapping around" event triggered by two cases. 1) z, r represent two positive integers, but $x = z + r$ represents a negative integer. 2) z, r represent two negative integers, but $x = z + r$ represents a positive integer due to the implicit modular operation within $\mathbb{F}_p$.

In the following subsection, we propose a novel truncation protocol that narrows the gap to just 1 bit. Concretely, the modified constraint $z \in [0, (p-1)/2]$ eliminates the "wrap around" event triggered by the first case, facilitating the detection and correction of the truncation errors of large magnitude. Further the truncation equation holds for any r (not just for uniform r with high probability). This enables us to choose smaller p (say with 32-bit length), thereby improving computation and communication efficiency.

3.2 Truncation with Only 1-Bit Gap

We introduce a novel truncation method in Theorem 3.2, which effectively reduces the size of the gap between the secret value and the actual modulus to just 1 bit. Notably, this gap size remains fixed and is independent of the correctness of the truncation result.

Theorem 3.2. *In the field $\mathbb{F}_p$ where $p = 2^\ell - 1$, let $z \in [0,(p-1)/2]$ and z can be divided into two field numbers $x = z + r \mod p$ and $y = p - r$ such that $x + y = z \mod p$ where $r \in \mathbb{F}_p$ can be any field element. Then we have $\text{Trunc}_d(x) + \text{Trunc}_d(y) + \delta(r,x) \cdot (2^{\ell-d} - 1) \in \text{Trunc}_d(z) + E$, where $\delta(r,x) = 1$ if $r_{msb} = 0 \wedge x_{msb} = 1$, and 0 otherwise.*

Proof. For $z, r \in \mathbb{F}_p$, define $z = z_1 \cdot 2^d + z_2$, we have $\text{Trunc}_d(z) = \lfloor z/2^d \rfloor = z_1$. Consider the following cases.
Case 1: $0 \le r \le (p-1)/2$, which means $r_{\text{msb}} = 0$. In this case, we define $r = r_1 \cdot 2^d + r_2$, where $0 \le r_2 < 2^d$. It is easy to know that, there exists some integers $c \in \{0,1\}$ and $0 \le \varepsilon \le 2^d - 1$ such that $r_2 + z_2 = c \cdot 2^d + \varepsilon$. By definition, $\text{Trunc}_d(y) = p - \lfloor r/2^d \rfloor = p - r_1$ because $(p-1)/2 < y < p$.

Since $0 \le z \le (p-1)/2$, we have $x = z + r = (z_1 + r_1) \cdot 2^d + (z_2 + r_2) \in \mathbb{F}_p$. We further discuss the following two subcases.

1. $0 \le x \le (p-1)/2$, which means $x_{\text{msb}} = 0$. In this case, we have

$$\begin{aligned}\text{Trunc}_d(x) &= \lfloor (z+r)/2^d \rfloor \\ &= \lfloor r_1 + z_1 + (r_2 + z_2)/2^d \rfloor \\ &= r_1 + z_1 + \lfloor (r_2 + z_2)/2^d \rfloor = r_1 + z_1 + c\end{aligned}$$

   Therefore, we have $\text{Trunc}_d(x) + \text{Trunc}_d(y) = r_1 + z_1 + c + (p - r_1) = z_1 + c = \text{Trunc}_d(z) + c \in \mathbb{F}_p$.

2. $(p-1)/2 < x \le p - 1$, which means $x_{\text{msb}} = 1$. In this case, we have

$$\begin{aligned}&\text{Trunc}_d(x) \\ &= p - \lfloor (p - (z+r))/2^d \rfloor \\ &= p - \lfloor (p - (z_1 \cdot 2^d + r_1 \cdot 2^d + z_2 + r_2))/2^d \rfloor \\ &= p - \lfloor (2^\ell - 1 - (z_1 + r_1) \cdot 2^d - (z_2 + r_2))/2^d \rfloor \\ &= p - \left\lfloor \frac{2^\ell - (z_1 + r_1 + c + 1) \cdot 2^d + (2^d - 1 - \varepsilon)}{2^d} \right\rfloor \\ &= p - (2^{\ell-d} - 1 - (z_1 + r_1 + c)) - \left\lfloor \frac{2^d - 1 - \varepsilon}{2^d} \right\rfloor \\ &= p - (2^{\ell-d} - 1) + (z_1 + r_1 + c)\end{aligned}$$

   The last equation holds because $0 \le 2^d - 1 - \varepsilon < 2^d$. Therefore, we have $\text{Trunc}_d(x) + \text{Trunc}_d(y) + (2^{\ell-d} - 1) = p + (z_1 + r_1 + c) + p - r_1 = z_1 + c = \text{Trunc}_d(z) + c \in \mathbb{F}_p$.

Case 2: $(p-1)/2 < r \le p - 1$, which means $r_{\text{msb}} = 1$. In this case, we have $0 < p - r \le (p-1)/2$. Define $p - r = \tilde{r}_1 \cdot 2^d + \tilde{r}_2$, where $0 \le \tilde{r}_2 < 2^d$. Therefore, $\text{Trunc}_d(y) = \lfloor (p-r)/2^d \rfloor = \tilde{r}_1$. We further discuss the following two subcases.

1. $0 \le x \le (p-1)/2$, which means $x_{\text{msb}} = 0$. In this case, define $\tilde{c} \in \{0,1\}$ such that $\tilde{c} = 0$ if $\tilde{r}_2 \le z_2$, and $\tilde{c} = 1$ otherwise. We have $x = z + r - p \in \mathbb{F}_p$, and

$$\begin{aligned}\text{Trunc}_d(x) &= \lfloor (z + r - p)/2^d \rfloor \\ &= \lfloor ((z_1 - \tilde{r}_1) \cdot 2^d + (z_2 - \tilde{r}_2))/2^d \rfloor \\ &= \lfloor ((z_1 - \tilde{r}_1 - \tilde{c}) \cdot 2^d + (z_2 - \tilde{r}_2 + \tilde{c} \cdot 2^d))/2^d \rfloor \\ &= z_1 - \tilde{r}_1 - \tilde{c}\end{aligned}$$

   The last equation holds because $0 \le z_2 - \tilde{r}_2 + \tilde{c} \cdot 2^d < 2^d$. Therefore, we have $\text{Trunc}_d(x) + \text{Trunc}_d(y) = z_1 - \tilde{r}_1 - \tilde{c} + \tilde{r}_1 = \text{Trunc}_d(z) - \tilde{c}$.

2. $(p-1)/2 < x \le p - 1$, which means $x_{\text{msb}} = 1$. In this case, define $\hat{c} \in \{0,1\}$ such that $\hat{c} = 0$ if $z_2 \le \tilde{r}_2$, and $\hat{c} = 1$ otherwise. We have $x = z + r \in \mathbb{F}_p$, and

$$\begin{aligned}\text{Trunc}_d(x) &= p - \lfloor (p - z - r)/2^d \rfloor \\ &= p - \lfloor (\tilde{r}_1 \cdot 2^d + \tilde{r}_2 - z_1 \cdot 2^d - z_2)/2^d \rfloor \\ &= p - \lfloor ((\tilde{r}_1 - z_1 - \hat{c}) \cdot 2^d + (\tilde{r}_2 - z_2 + \hat{c} \cdot 2^d))/2^d \rfloor \\ &= p - \tilde{r}_1 + z_1 + \hat{c}\end{aligned}$$

   The last equation holds because $0 \le \tilde{r}_2 - z_2 + \hat{c} \cdot 2^d < 2^d$. Therefore, we have $\text{Trunc}_d(x) + \text{Trunc}_d(y) = p - \tilde{r}_1 + z_1 + \hat{c} + \tilde{r}_1 = \text{Trunc}_d(z) + \hat{c} \in \mathbb{F}_p$.

This concludes the proof. □

It is worth noting that Theorem 3.2 can be extended to a power-of-two ring $\mathbb{Z}_{2^\ell}$ in a natural way by replacing the Mersenne prime p with 2^ℓ. It allows the truncation protocol of [12, 20] in $\mathbb{Z}_{2^\ell}$ to be realized in a simpler manner.

Dealing with Negative Integers. Theorem 3.2 imposes the contraint $z \in [0,(p-1)/2]$ on the truncated value z. In fact, it is possible to support truncation on negative integers. The following Corollary 3.3 shows how to handle field elements that represent negative integers.

Corollary 3.3. *In the field $\mathbb{F}_p$ where $p = 2^\ell - 1$, let $a \in [0, 2^{\ell-2}) \cup [p - 2^{\ell-2}, p)$ and $b = a + 2^{\ell-2}$ such that $b \in [0,(p-1)/2]$. We have $\text{Trunc}_d(b) - 2^{\ell-d-2} \in \text{Trunc}_d(a) + E$.*

Proof. Let $z = b$, $r = -2^{\ell-2}$ and $x = z + r = a$. According to Theorem 3.2, we have $\text{Trunc}_d(a) + \text{Trunc}_d(2^{\ell-2}) \in \text{Trunc}_d(b) + E$, because $r_{\text{msb}} = 1$. The proof follows since $\text{Trunc}_d(2^{\ell-2}) = 2^{\ell-d-2}$. □

Protocol 3.1. Π_{Trunc} *(Newly added!)*

Input: $[a]_p$ where $a \in [-2^{\ell-2}, 2^{\ell-2})$.
Output: $[f]_p \in [\text{Trunc}_d(a)]_p + 2E$.
Common Randomness: $([r]_p, [r]_{\text{B}}) \leftarrow \mathcal{F}_{\text{SolvedBits}}$.

1. $[r_{\text{msb}}]_p \leftarrow [r_{\ell-1}]_p$
2. $[r']_p \leftarrow \sum_{i=d}^{\ell-1} 2^{i-d} \cdot [r_i]_p + \sum_{i=\ell-d}^{\ell-1} 2^i \cdot [r_{\text{msb}}]_p$
3. $[b]_p \leftarrow [a]_p + 2^{\ell-2}$ such that $b \in [0, (p-1)/2]$
4. $\langle c \rangle_p \leftarrow [b]_p + [r]_p$
5. $c \leftarrow \Pi_{\text{Reveal}}\ ([c]_p)$, and compute $\text{Trunc}_d(c)$ and c_{msb}
6. $[e]_p \leftarrow (1 - [r_{\text{msb}}]_p) \cdot c_{\text{msb}}$
7. $[f]_p \leftarrow \text{Trunc}_d(c) - [r']_p + [e]_p \cdot (2^{\ell-d} - 1)$
8. Output $[f]_p - 2^{\ell-d-2}$

As a result, we are able to perform truncation operation over a signed integer $a \in \left[0, 2^{\ell-2}\right) \cup \left[p - 2^{\ell-2}, p\right)$, described in Protocol Π_{Trunc}. Note that $[r_{\text{msb}}]_p$ and $[r']_p$ computed in Step 1 and Step 2 can be generated in the preprocessing phase. More generally, we can think of $([r']_p, [r]_p, [r_{\text{msb}}]_p)$ as a random truncation triple consumed for every pure truncation operation over secret integers.

Theorem 3.4. Π_{Trunc} *securely realizes* $\mathcal{F}_{\text{Trunc}}$ *in the* $\mathcal{F}_{\text{SolvedBits}}$*-hybrid model with abort, in the presence of a fully semi-honest adversary controlling t corrupted parties.*

Proof. The correctness follows immediately from Theorem 3.2 and Corollary 3.3. Step 3 and Step 8 are applying Corollary 3.3 to a, then one gets $\text{Trunc}_d(a) \in \text{Trunc}_d(b) - 2^{\ell-d-2} + E$. Steps 4 - 7 are applying Theorem 3.2 to b under the condition of $b = a + 2^{\ell-2} \in [0, (p-1)/2]$, then it holds that $\text{Trunc}_d(b) \in \text{Trunc}_d(b+r) - \text{Trunc}_d(r) + \delta(r, b+r) \cdot (2^{\ell-d} - 1) + E$ for some random r, where $b + r$ can be revealed as c in Step 5 and $\delta(r, b+r)$ can be evaluated locally as e in Step 6.

As for privacy, note that the only leaked information is $c = b + r$ in Step 5, where b is relevant to a. Since r is assumed to be a uniformly random, unknown value from $\mathbb{F}_p$, independent of a and b, it follows that c is uniformly random in $\mathbb{F}_p$ and leaks no information about a. □

3.3 Fixed-Point Multiplication

We further extend the truncation process above to build a 1-round fixed-point multiplication protocol, as described in Protocol $\Pi_{\text{Fixed-Mult}}$. Assuming we have another kind of random truncation triple $([r']_p, \langle r \rangle_p, [r_{\text{msb}}]_p)$ where r is assumed a uniformly random value in $\mathbb{F}_p$ and $r' = \text{Trunc}_d(r)$, we can realize it with the same online complexity as DN multiplication. Note that the second component is a degree-$2t$ sharing. We defer the detailed protocol $\Pi_{\text{Trunc-Triple}}$ of generating such a random truncation triple to Section 3.4.

Protocol 3.2. $\Pi_{\text{Fixed-Mult}}$

Input: $[a]_p$ and $[b]_p$.
Output: $[y]_p \in [\text{Trunc}_d(a \cdot b)]_p + 2E$.
Common Randomness: A random truncation triple $([r']_p, \langle r \rangle_p, [r_{\text{msb}}]_p) \leftarrow \mathcal{F}_{\text{Trunc-Triple}}$ such that $r' = \text{Trunc}_d(r) \in \mathbb{F}_p$ and $r_{\text{msb}} \in \{0, 1\}$ is the MSB of r.

1. $\langle c \rangle_p \leftarrow [a]_p \cdot [b]_p + \langle r \rangle_p + 2^{\ell-2}$
2. $c \leftarrow \Pi_{\text{Reveal}}\ (\langle c \rangle_p)$, and compute $\text{Trunc}_d(c)$ and c_{msb}
3. $[e]_p \leftarrow (1 - [r_{\text{msb}}]_p) \cdot c_{\text{msb}}$
4. $[f]_p \leftarrow \text{Trunc}_d(c) - [r']_p + [e]_p \cdot (2^{\ell-d} - 1) - 2^{\ell-d-2}$
5. Output $[f]_p$

Before digging into the fixed-point multiplication protocol, it is necessary to discuss the translation from a fixed-point number to a field element. Considering a fixed-point number x in which the bit length is ℓ and the size of the fractioanl part is d such that $x = \sum_{i=0}^{\ell-1} x_i \cdot 2^{i-d}$ where x_i is the i-th bit of x, we can represent x with a corresponding field element by computing $a = x \cdot 2^d \in \mathbb{F}_p$. Similarly, we can represent a fixed-point number y with $b = y \cdot 2^d \in \mathbb{F}_p$. Then multiplying a and b yields $ab = xy \cdot 2^{2d}$ after which a truncation operation is required to obtain $\text{Trunc}_d(ab) = xy \cdot 2^d$ which preserves the original fractional precision.

Back to our fixed-point multiplication protocol described in Protocol $\Pi_{\text{Fixed-Mult}}$, the inputs are $[a]_p$ and $[b]_p$ representing two fixed-point numbers. In the online phase, only Step 2 requires communication of 2 field elements per party, which is the same as DN multiplication. Hence, the online complexity is exactly 1 round with 2 field elements per party. The random truncation triple can be generated in the preprocessing phase, with complexity of 2 rounds and 3ℓ field elements per party (deferred to be described in Section 3.4).

Theorem 3.5. $\Pi_{\text{Fixed-Mult}}$ *securely realizes* $\mathcal{F}_{\text{Fixed-Mult}}$ *in the* $\mathcal{F}_{\text{Trunc-Triple}}$*-hybrid model with abort, in the presence of a fully semi-honest adversary controlling t corrupted parties.*

Proof. The correctness can be proved similarly as Theorem 3.4 where ab is the value to be truncated. As for privacy, the value c revealed in Step 2 leaks no information about a or b. The sharings $[a]_p$ and $[b]_p$ are degree-t sharings, so $[a]_p[b]_p$ is a degree-$2t$ sharing of ab. Therefore, $\langle c \rangle_p = [a]_p[b]_p + \langle r \rangle_p + 2^{\ell-2}$ is a uniformly random degree-$2t$ sharing of $ab + r + 2^{\ell-2}$, as r is assumed to be uniformly random. □

3.4 Generation of Random Truncation Triple

In this section, we introduce a simple method to generate the random truncation triple $([r']_p, \langle r \rangle_p, [r_{\text{msb}}]_p)$ consumed in the fixed-point multiplication protocol, where $r \in \mathbb{F}_p$ is uniformly random, $r' = \text{Trunc}_d(r) \in \mathbb{F}_p$ and r_{msb} is the MSB of r.

Protocol 3.3. $\Pi_{\text{Trunc-Triple}}$

Input: None.
Output: $([r']_p, \langle r \rangle_p, [r_{\text{msb}}]_p)$, where $r' = \text{Trunc}_d(r)$.

```
1. $([r]_p, [r]_B) \leftarrow \mathcal{F}_{\text{SolvedBits}}$
2. $[r_{\text{msb}}]_p \leftarrow [r_{\ell-1}]_p$.
3. $[r']_p \leftarrow \sum_{i=d}^{\ell-1} 2^{i-d} \cdot [r_i]_p + \sum_{i=\ell-d}^{\ell-1} 2^i \cdot [r_{\text{msb}}]_p$
4. $\langle r \rangle_p \leftarrow \sum_{i=0}^{\ell-1} 2^i \cdot [r_i]_p \cdot [r_i]_p + \langle 0 \rangle_p$
```

As described in Protocol $\Pi_{\text{Trunc-Triple}}$, we utilize the functionality $\mathcal{F}_{\text{SolvedBits}}$ to generate ℓ random bits $\{r_i\}_{i=0}^{\ell-1}$ and the corresponding random element $r = \sum_{i=0}^{\ell-1} 2^i \cdot r_i$ as the raw material of such a random truncation triple. Note that all the steps can be done in the preprocessing phase and the complexity of this protocol depends on the generation of the solved bits, that is 2 rounds and 3ℓ field elements per party.

Theorem 3.6. $\Pi_{\text{Trunc-Triple}}$ *securely realizes* $\mathcal{F}_{\text{Trunc-Triple}}$ *in the* $\mathcal{F}_{\text{SolvedBits}}$*- hybrid model with abort, in the presence of a fully semi-honest adversary controlling t corrupted parties.*

Proof. The correctness of Step 3 follows directly from Theorem 3.1 which tells us that the truncation operation can be done at bit-level, i.e. shift the bits of r down by d positions and fill the top d bits with the MSB of r. In Step 4, it holds that $r = \sum_{i=0}^{\ell-1} 2^i \cdot r_i \cdot r_i$ since $r_i \in \{0,1\}$. Note that we use the pseudo-random sharing of zero (PRZS [11]) technique to randomize the polynomial coefficients of $\langle r \rangle_p$ where the pseudo-random degree-$2t$ sharing $\langle 0 \rangle_p$ can be generated by PRG without communication.

The privacy follows from the fact that we only invoke the private ideal functionality $\mathcal{F}_{\text{SolvedBits}}$. □

4 Improved Bitwise Primitives

Before introducing the protocols of building blocks in privacy-preserving machine learning, we first propose our novel method to compute the prefix-ORs, which is the core and bottleneck to realizing the bitwise less-than functionality.

4.1 Prefix-ORs Protocol

In this section, we consider prefix-ORs, which computes $b_j = \bigvee_{i=1}^{j} a_i$ for $j = 1, \ldots, \ell$ where $a_i \in \{0,1\}$. We propose a simple and efficient method to securely compute prefix-OR, as described in Protocol Π_{PreOR}. Intuitively, instead of computing prefix-ORs, we compute a dual problem. I.e., $\overline{b}_j = \bigwedge_{i=1}^{j} \bar{a}_i$ for $j = 1, \ldots, \ell$.

Note that this protocol only composes one invocation of Π_{PreMult} while other steps can be done locally. Hence, the online complexity is 1 round with 2ℓ field elements per party and the preprocessing complexity is 2 rounds with 7ℓ field elements per party.

Protocol 4.1. Π_{PreOR}

Input: $[a_1]_p, [a_2]_p, \ldots, [a_\ell]_p$ where $a_i \in \{0,1\}$.
Output: All prefix ORs $[a_1]_p, [a_1 \vee a_2]_p, \ldots, [\vee_{i=1}^{\ell} a_i]_p$.

```
1. For $i = 1, \ldots, \ell$: compute $[b_i]_p \leftarrow 1 - [a_i]_p$.
2. $([c_1]_p, \ldots, [c_\ell]_p) \leftarrow \mathcal{F}_{\text{PreMult}}([b_1]_p, \ldots, [b_\ell]_p)$
3. Output $(1 - [c_1]_p, \ldots, 1 - [c_\ell]_p)$.
```

Protocol 4.2. $\Pi_{\text{Bitwise-LT}}$

Input: a and $[b]_B$ where $a, b \in \mathbb{F}_p$ and $b = \sum_{i=0}^{\ell-1} 2^i b_i$
Output: $[a < b]_p$

```
1. Let $a_0, \ldots, a_{\ell-1}$ be the decomposed bits of $a$.
2. Set $\bar{a}_i = 1 - a_i$ and $\bar{b}_i = 1 - [b_i]_p$ for $0 \le i \le \ell - 1$.
3. For $i = 0, \ldots, \ell - 1$: compute $[c_i]_p = \bar{a}_i \oplus [\bar{b}_i]_p$.
4. Run $([d_{\ell-1}]_p, \ldots, [d_0]_p) \leftarrow \mathcal{F}_{\text{PreOR}}([c_{\ell-1}]_p, \ldots, [c_0]_p)$
5. Set $[e_i]_p = [d_i - d_{i+1}]_p$ where $[e_{\ell-1}]_p = [d_{\ell-1}]_p$
6. Output $[a < b]_p = \sum_{i=0}^{\ell-1} \bar{a}_i \cdot [e_i]_p$.
```

Theorem 4.1. Π_{PreOR} *securely realizes* $\mathcal{F}_{\text{PreOR}}$ *in the* $\mathcal{F}_{\text{PreMult}}$*-hybrid model with abort, in the presence of a fully semi-honest adversary controlling t corrupted parties.*

Proof. The correctness holds because computing $b_j = \bigvee_{i=1}^{j} a_i$ is equivalent to copmute $\overline{b}_j = \bigwedge_{i=1}^{j} \bar{a}_i$ where $\bar{a}_i$ denote the opposite bit of a_i and so is $\overline{b}_j$. The privacy follows from the fact that this protocol only invokes the private ideal functionality $\mathcal{F}_{\text{PreMult}}$. □

4.2 Bitwise Less Than Protocol

Based on the above optimized Prefix-ORs protocol, we propose a round-efficient protocol to securely compute the bitwise less-than circuit. Specifically, given a public value a and shared bits $[b]_B$, it outputs the share of $(a < b)$. Thanks to the special form of Mersenne prime, in the online phase, the round complexity of the resulting protocol is just 1 round and the communication complexity is 2ℓ field elements per party. The preprocessing complexity is 2 rounds and 7ℓ field elements per party.

Our protocol described in Protocol $\Pi_{\text{Bitwise-LT}}$ follows the method in [15], with the substitution of our optimized prefix-OR subprotocol. In order to further reduce round complexity, we equivalently compute $p - b < p - a$, because $p - a$ is public. A key observation is that given the bit representation of $b = (b_0, \ldots, b_{\ell-1})$, the bit representation of $p - b$ is simply $(1 - b_0, \ldots, 1 - b_{\ell-1})$. This is because p is a Mersenne prime, and its bit representation is $(1, \ldots, 1)$.

Theorem 4.2. $\Pi_{\text{Bitwise-LT}}$ *securely realizes in the* $\mathcal{F}_{\text{PreOR}}$*-hybrid model with abort, in the presence of a fully semi-honest adversary controlling t corrupted parties.*

Proof. Note that a, b are in the range of $[0, p-1]$. Considering $\bar{a}, \bar{b} \in \mathbb{Z}_{2^\ell}$ where ℓ is the bit-length of p, define $\bar{a} = p - a$ and $\bar{b} = p - b$. Observe that $a < b$ if and only if $\bar{b} < \bar{a}$. One can compute $[\bar{b}]_{\text{B}} = (1 - [b_0]_p, \ldots, 1 - [b_{\ell-1}]_p)$ as described in Step 2. This is because $p = 2^\ell - 1$ is in the form of all 1s.

In the following steps, we compare $\bar{b} < \bar{a}$. We start from the most significant bit and look for the first bit where $\bar{b}$ and $\bar{a}$ differ. The XOR in Step 3 computes a bitwise sharing $[c]_{\text{B}}$ with ones on all positions where the bits differ. In Step 4, we invoke $\mathcal{F}_{\text{PreOR}}$ to compute all prefix-`OR`s starting from MSB. This results in a string $(d_{\ell-1}, \ldots, d_0)$ which contains a series of 0's followed by 1's starting at the first location where $\bar{b}$ and $\bar{a}$ differ, denoted by k. As computed in Step 5, we have $e_i = 1$ if $i = k$, and $e_i = 0$ otherwise. In the special case $\bar{b} = \bar{a}$, all $e_i = 0$ so the output is clear 0 as it should be. Otherwise, the output is $\bar{a}_k$ that can be extracted with $e_k = 1$.

The privacy follows from the fact that the protocol only invokes the private ideal functionality $\mathcal{F}_{\text{PreOR}}$. $\square$

5 Building Blocks in Neural Network

In this section, we first describe an efficient way to realize DReLU in a field over Mersenne prime. Then we use the state-of-the-art technique in [22] (CRYPTO 2021) to realize ReLU and Max (the unit operation in Maxpool) with the same round complexity as DReLU.

5.1 DReLU

ReLU is a non-linear activation function that is commonly used in neural networks, which can solve the vanishing gradient problem. It can be represented as $\text{ReLU}(x) = \text{Max}(0, x)$ and the derivative of ReLU (DReLU) is 1 for $x \geq 0$ and 0 for $x < 0$. The DReLU function on the input of integers is defined as follows, and $\text{ReLU}(x) = \text{DReLU}(x) \cdot x$.

$$\text{DReLU}(x) = \begin{cases} 1 & , x \geq 0 \\ 0 & , x < 0 \end{cases}$$

As described in Section 3.1, the most significant bit indicates the sign of the underlying integers represented in the field of Mersenne prime, which means DReLU is related to extracting the most significant of its representation in the field. It is observed in [44] that $a_{\text{msb}} = (2a)_{\text{lsb}}$ if the multiplication is done over an odd ring. Hence, it holds that $\text{DReLU}(a) = 1 - (2a)_{\text{lsb}}$.

As in [44], in order to compute $[(2a)_{\text{lsb}}]_p$, we first mask $2a$ with a uniform r such that $y = 2a + r$ and then reveal y. Note that $y_{\text{lsb}} = (2a)_{\text{lsb}} \oplus r_{\text{lsb}}$ if $y \geq r$, and $y_{\text{lsb}} = (2a)_{\text{lsb}} \oplus r_{\text{lsb}} \oplus 1$ otherwise. Given the bits sharing $[r]_{\text{B}}$, we are able to decide whether $2a + r$ wraps around the field by $\Pi_{\text{Bitwise-LT}}$ described in Section 4.2. It holds that $(2a)_{\text{lsb}} = y_{\text{lsb}} \oplus r_{\text{lsb}} \oplus (y < r)$.

Protocol 5.1. Π_{DReLU}

Input: $[a]_p$ where $a \in \mathbb{F}_p$
Output: $[\text{DReLU}(a)]_p$
Common Randomness: $([r]_p, [r]_{\text{B}}) \leftarrow \mathcal{F}_{\text{SolvedBits}}$.

1. $[y]_p \leftarrow [2a + r]_p$ and reveal $y \leftarrow \Pi_{\text{Reveal}}([y]_p)$.
2. $[b]_p \leftarrow y_{\text{lsb}} \oplus [r_0]_p$.
3. Run $[c]_p \leftarrow \mathcal{F}_{\text{Bitwise-LT}}(y, [r]_{\text{B}})$
4. Output $1 - [b]_p \oplus [c]_p$

The detailed protocol is described in Protocol Π_{DReLU}. Note that Step 4 can be realized by a DN multiplication. In the online phase, the complexity is 3 rounds with $4 + 2\ell$ field elements per party (including approximately 2 for Π_{Reveal}, 2ℓ for $\Pi_{\text{Bitwise-LT}}$, and 2 for Π_{Mult}). As for the preprocessing phase, all random values can be generated in 2 rounds with $1 + 10\ell$ field elements per party (including 3ℓ for $\Pi_{\text{SolvedBits}}$, 7ℓ for $\Pi_{\text{Bitwise-LT}}$, and 1 for Π_{Mult}).

Theorem 5.1. Π_{DReLU} *securely realizes* $\mathcal{F}_{\text{DReLU}}$ *in the* $(\mathcal{F}_{\text{Bitwise-LT}}, \mathcal{F}_{\text{Mult}})$*-hybrid model with abort, in the presence of a fully semi-honest adversary controlling t corrupted parties.*

Proof. The correctness follows easily from $(2a)_{\text{lsb}} = y_{\text{lsb}} \oplus r_{\text{lsb}} \oplus (y < r)$. As for privacy, the revealed value $y = 2a + r$ in Step 1 leaks no information about a, as r is uniformly random. Then the invocation of the ideal functionality $\mathcal{F}_{\text{Bitwise-LT}}$ is private. $\square$

5.2 Vectorized Two-Layer DN Multiplication

In the context of neural networks, several operators are derived from the DReLU operation. One crucial observation is that the final step of the Π_{DReLU} protocol involves a multiplication. Additionally, other operators like ReLU and Maxpool can be seen as a combination of a two-layer multiplication, along with DReLU. In this subsection, we extend and simplify the techniques introduced in [22] to efficiently compute vectorized two-layer multiplication with a single round of online complexity.

The vectorized two-layer multiplication protocol aims to produce the outputs $\{[xyz_i]_p\}_{i=1}^m$ given the inputs $[x]_p$, $[y]_p$, and $\{[z_i]_p\}_{i=1}^m$. The basic idea behind this protocol is as follows:

1. In the standard DN multiplication protocol, when computing $[xy]_p$, each party receives a value $u = xy + r$, where r is a random value.

2. We can utilize the fact that $0 = z_i + (-z_i)$, where z_i is a known input value. If we can prepare Beaver triples

Protocol 5.2. $\Pi_{\text{2L-DN}}$

Input: $[x]_p, [y]_p$ and $\{[z_i]_p\}_{i=1}^m$
Output: $\{[xyz_i]_p\}_{i=1}^m$
Common Randomness: $m+1$ pairs of double sharings $([r]_p, \langle r\rangle_p)$ and $\{([r_i]_p, \langle r_i\rangle_p)\}_{i=1}^m$.

1. Compute $\langle u\rangle_p = [x]_p \cdot [y]_p + \langle r\rangle_p$.
   Compute $\langle u_i\rangle_p = [r]_p \cdot [-z_i]_p + \langle r_i\rangle_p$.
2. Reveal $u \leftarrow \Pi_{\text{Reveal}}\ (\langle u\rangle_p)$ and $u_i \leftarrow \Pi_{\text{Reveal}}\ (\langle u_i\rangle_p)$.
3. Compute $[c_i]_p = u_i - [r_i]_p$ for $1 \le i \le m$.
4. For $i = 1, \ldots, m$, locally compute:
   $[xyz_i]_p = u \cdot [z_i]_p + [c_i]_p$

$([r]_p, [-z_i]_p, [-rz_i]_p)$ for $1 \le i \le m$, then it becomes possible to compute $[xyz_i]_p$ locally for each $1 \le i \le m$.

3. It is worth noting that computing $[rz_i]_p$ and $[xy]_p$ can be carried out in parallel, resulting in a 1-round online complexity.

By following this approach, we can effectively compute the vectorized two-layer multiplication protocol with improved efficiency, reducing the online complexity to just one round with $2(m+1)$ field elements per party. In the preprocessing phase, it generates $m+1$ pairs of double sharings with communication of $m+1$ field elements per party.

Theorem 5.2. $\Pi_{\text{2L-DN}}$ *securely realizes* $\mathcal{F}_{\text{2L-DN}}$ *in the* $\mathcal{F}_{\text{Mult}}$*-hybrid model with abort, in the presence of a fully semi-honest adversary controlling t corrupted parties.*

Proof. According to the protocol, we have $u = xy + r$ and $u_i = -rz_i + r_i$ for $1 \le i \le m$. Therefore $c_i = -rz_i$ for $1 \le i \le m$. Since u is public and $u \cdot z_i + c_i = (xy+r)\cdot z_i + c_i = xyz_i$, then Step 4 holds. The privacy follows from the fact that the protocol parallelly invokes the ideal functionality $\mathcal{F}_{\text{Mult}}$. □

5.3 ReLU

We now use our vectorized two-layer multiplication protocol to realize ReLU. Note that $\text{ReLU}(a) = \text{DReLU}(a) \cdot a$. The last step of $\text{DReLU}(a)$ is $1 - b \oplus c$, which can be rewritten as $1 - (b-c)\cdot(b-c)$ for $b, c \in \{0,1\}$. Therefore, $\text{ReLU}(a) = a - a(b-c)(b-c)$, which could be realized using our two-layer multiplication protocol. The online complexity of this protocol is 3 rounds with $6 + 2\ell$ field elements per party (including 2 for Π_{Reveal}, 2ℓ for $\Pi_{\text{Bitwise-LT}}$, and 4 for $\Pi_{\text{2L-DN}}$ where $m = 1$). Details are given in Protocol Π_{ReLU}.

As for the preprocessing phase, all random values can be generated in 2 rounds with $2 + 10\ell$ field elements per party (including 3ℓ for $\Pi_{\text{SolvedBits}}$, 7ℓ for $\Pi_{\text{Bitwise-LT}}$, and 2 for $\Pi_{\text{2L-DN}}$).

Protocol 5.3. Π_{ReLU}

Input: $[a]_p$ where $a \in \mathbb{F}_p$
Output: $[\text{ReLU}(a)]_p$
Common Randomness: $([r]_p, [r]_{\text{B}}) \leftarrow \mathcal{F}_{\text{SolvedBits}}$.

1. $[y]_p \leftarrow [2a + r]_p$ and reveal $y \leftarrow \Pi_{\text{Reveal}}\ ([y]_p)$.
2. $[b]_p \leftarrow y_{\text{lsb}} \oplus [r_0]_p$.
3. Run $[c]_p \leftarrow \mathcal{F}_{\text{Bitwise-LT}}\ (y, [r]_{\text{B}})$
4. Run $[t]_p \leftarrow \mathcal{F}_{\text{2L-DN}}\ ([b-c]_p, [b-c]_p, \{[a]_p\})$.
5. Output $[a]_p - [t]_p$.

Theorem 5.3. Π_{ReLU} *securely realizes* $\mathcal{F}_{\text{ReLU}}$ *in the* ($\mathcal{F}_{\text{Bitwise-LT}}$, $\mathcal{F}_{\text{2L-DN}}$)*-hybrid model with abort, in the presence of a fully semi-honest adversary controlling t corrupted parties.*

Proof. As for the correctness, we have $\text{DReLU}(a) = 1 - b \oplus c = 1 - (b-c)\cdot(b-c)$ as described in Step 4 of Π_{DReLU}. Hence, it holds that $\text{ReLU}(a) = \text{DReLU}(a)\cdot a = a - (b-c)\cdot(b-c)\cdot a$. The correctness holds due to the correctness of Π_{DReLU} and $\Pi_{\text{2L-DN}}$.

As for privacy, the revealed value $y = 2a + r$ in Step 1 leaks no information about a, as r is uniformly random. Then the following invocations of the ideal functionalities $\mathcal{F}_{\text{Bitwise-LT}}$ and $\mathcal{F}_{\text{2L-DN}}$ are private. □

5.4 Maxpool

Maxpool is the key component in convolutional neural networks (CNN). When they operate on a vector of m elements, Maxpool returns the maximum element. Assume that m is the power of two for simplicity.

The crucial operation is the comparison that can be computed with DReLU. It is easily to check that $(a < b) = 1 - \text{DReLU}(a-b)$. We can represent $\text{Max}(a,b)$ as follows, that is b if $a < b$ and a otherwise.

$$\begin{aligned}\text{Max}(a,b) &= \text{DReLU}(a-b)\cdot a + [1 - \text{DReLU}(a-b)]\cdot b \\ &= \text{ReLU}(a-b) + b\end{aligned}$$

Hence, it can be realized with the same complexity as Π_{ReLU}.

To find the maximum value of m elements, we first group the values into pairs and compare each pair to obtain the maximum of two, resulting in a vector of size $m/2$. This process can be repeated for $\log m$ times. Details are given in Protocol Π_{Maxpool}.

It consists of $\log m$ rounds and $m-1$ invocations of ReLU, which results in the online complexity of $3\log m$ rounds with $(m-1)\cdot(6+2\ell)$ field elements per party. As for the preprocessing phase, all random values can be generated in 2 rounds with $(m-1)\cdot(2+10\ell)$ field elements per party.

```
Protocol 5.4. $\Pi_{\text{Maxpool}}$

Input: $[a_1]_p, \dots, [a_m]_p$ assuming $m$ is powers of two.
Output: $[a_k]_p$ where $a_k = \text{Maxpool}(a_1, \dots, a_m)$.
 1. If $m = 1$: Output $[a_1]_p$
 2. Set $\boldsymbol{a}^1 \leftarrow ([a_1]_p, \dots, [a_m]_p)$ and $s \leftarrow m$.
 3. For $i = 1, \dots, \log m$ :
     a. $s \leftarrow s/2$.
     b. Set $([b_1]_p, \dots, [b_s]_p, [c_1]_p, \dots, [c_s]_p) \leftarrow \boldsymbol{a}^i$.
     c. For $j = 1, \dots, s$: $[e_j]_p \leftarrow \mathcal{F}_{\text{ReLU}}([b_j - c_j]_p) + c_j$.
     d. If $s = 1$: Output $[e_1]_p$.
        Else: Set $\boldsymbol{a}^{i+1} \leftarrow ([e_1]_p, \dots, [e_l]_p)$
```

Theorem 5.4. Π_{Maxpool} *securely realizes* $\mathcal{F}_{\text{Maxpool}}$ *in the* $\mathcal{F}_{\text{ReLU}}$*-hybrid model with abort, in the presence of a fully semi-honest adversary controlling t corrupted parties.*

Proof. We have $\text{Max}(b_j, c_j) = \text{ReLU}(b_j - c_j) + c_j$ as computed in Step c. The process can be described in a binary tree of depth $\log m$. Each node (except the leaf nodes) performs the Max operation on two child nodes and the root node outputs the maximum value. The privacy follows from the fact that the protocol only parallelly invokes $\mathcal{F}_{\text{ReLU}}$. □

6 Evaluations

We first summarize the theoretical round and communication complexity in Table 1. The round complexity including the online and preprocessing phases is independent of ℓ. Then we evaluate the performance of oblivious inference on 3 networks using MNIST. A number of prior works [24, 31, 34, 35, 42, 44, 45] evaluate over these networks.

Protocols	Rounds		Communication	
	Online	Prep.	Online	Prep.
$\Pi_{\text{Fixed-Mult}}$	1	2	2	3ℓ
Π_{PreMult}	1	2	2ℓ	7ℓ
Π_{PreOR}	1	2	2ℓ	7ℓ
$\Pi_{\text{Bitwise-LT}}$	1	2	2ℓ	7ℓ
Π_{DReLU}	3	2	$4+2\ell$	$1+10\ell$
$\Pi_{\text{2L-DN}}$	1	1	$2(m+1)$	$m+1$
Π_{ReLU}	3	2	$6+2\ell$	$2+10\ell$
Π_{Maxpool}	$3\log m$	2	$(m-1)(6+2\ell)$	$(m-1)(2+10\ell)$

Table 1: Round and communication complexity of building blocks. Numbers of communication are reported in field elements per party.

6.1 Experimental Setup

We have implemented this framework using approximately 15.4k lines of code (LOC) in C++. Our implementation leverages the communication backend of MP-SPDZ [25] and the neural network frontend of Falcon [45]. We conducted performance evaluations for inference in various settings involving different numbers of parties, including 3-party computation (3PC), 7PC, 11PC, 21PC, 31PC, and 63PC. Our experiments were conducted on a total of 11 servers, each of which was equipped with an AMD EPYC 7B12 64-core processor with 512GB of RAM.

To simulate different network conditions, we utilized the Linux `tc` command. We considered two network settings: a Local Area Network (LAN) setting with a shared 15 Gbps connection and an average Round-Trip Time (RTT) latency of approximately 0.3ms, and a Wide Area Network (WAN) setting with an RTT latency of 40ms and a maximum throughput of 100Mbps. In order to ensure reliable results, we collected data from 10 independent runs for each data point in our experiments, and the reported results are presented as the average values obtained from these runs.

In our implementation, all arithmetic operations on the shares are performed modulo $p = 2^{31} - 1$, with a fixed-point precision of 12 bits. These operations are carried out on 32-bit integers. Thanks to the 1-bit gap of the truncation protocol, we can use such a small prime for the underlying protocol. This enables us to get improved communication and round complexity without losing accuracy. To accelerate matrix multiplication, we utilize the Eigen library [23], which supports faster computation on our custom field type. The computations are parallelized using 16 cores to optimize Eigen's algorithm. Additionally, the servers employ hardware-accelerated AES-NI instructions for efficient random number generation. Furthermore, we incorporate an optimization technique inspired by the implementation of Falcon [45]. Specifically, we swap the order of the ReLU layer and the subsequent Maxpool layer to enhance performance. Overall, these optimizations and hardware acceleration techniques contribute to the efficiency and speed of our implementation.

Networks and Datasets. We employ the widely used MNIST dataset [46], which consists of a collection of 28×28 pixel images depicting handwritten numbers. The objective is to accurately predict the corresponding number for each image. We select three standard neural networks from the field of privacy-preserving machine learning. Network-A is a 3-layer DNN network derived from SecureML [35]. Network-B is a 3-layer CNN network derived from Chameleon [42]. Network-C is a 4-layer CNN network derived from MiniONN [31]. For more specific information and details about these neural networks, please refer to Appendix E.

6.2 Performance of Oblivious Inference

Table 2 and Table 3 present the specific performance results for evaluating different networks in the LAN and WAN settings, respectively. Furthermore, Table 4 provides the commu-

#PC	Network-A		Network-B		Network-C	
	Online	Prep.	Online	Prep.	Online	Prep.
3PC	0.006	0.006	0.007	0.02	0.02	0.178
7PC	0.01	0.008	0.01	0.027	0.03	0.254
11PC	0.011	0.01	0.011	0.038	0.033	0.335
21PC	0.008	0.017	0.01	0.059	0.035	0.508
31PC	0.011	0.023	0.013	0.077	0.047	0.632
63PC	0.024	0.048	0.025	0.149	0.096	1.208

Table 2: Online and preprocessing time for oblivious inference in the LAN setting. All numbers are reported in seconds.

#PC	Network-A		Network-B		Network-C	
	Online	Prep.	Online	Prep.	Online	Prep.
3PC	0.37	0.244	0.387	0.392	1.222	2.189
7PC	0.373	0.213	0.404	0.576	1.327	2.768
11PC	0.376	0.216	0.41	0.414	1.346	2.777
21PC	0.386	0.233	0.444	0.57	1.718	4.812
31PC	0.398	0.268	0.478	0.631	2.027	6.65
63PC	0.478	0.52	0.68	1.885	4.573	10.69

Table 3: Online and preprocessing time for oblivious inference in the WAN setting. All numbers are reported in seconds.

nication size observed during the evaluation of these networks with varying numbers of parties. The performance evaluation focuses on a single batch inference.

For 63 parties, the online (preprocessing) running time of evaluating Network C is only $0.1(1.21)$ seconds and $4.6(10.7)$ seconds in the LAN and WAN setting, respectively. The online and preprocessing communication cost of evaluating Network C are 2.82 MB and 18.3 MB respectively. As far as we know, our protocol is the first fully implemented PPML protocol that can support up to 63 parties.

Analysis of the Online Performance. We have observed that in the LAN setting, the time required for computation is directly proportional to the number of parties involved. This is due to the fact that the party responsible for collecting shares of a degree-$2t$ (or degree-t) sharing needs to perform the reconstruction of the secret and send it back to the other parties. As the number of shares to be collected increases, the computation cost also increases, thus impacting the overall inference performance.

On the other hand, in the WAN setting, we have noticed that the online time remains relatively consistent across different numbers of parties. In this scenario, the round complexity becomes the dominant factor affecting the inference performance rather than the computation cost.

Comparison to Prior Work In the honest-majority setting, most PPML protocols primarily focus on the three-party case [8, 12, 29, 34, 39, 44, 45] and the four-party case [4, 9, 13, 28]. These protocols often face challenges when extending to a larger number of parties. While general MPC protocols like [10, 14, 16, 18, 26, 27] can be used, they are not scalable, particularly in the context of PPML with a large number of parties.

#PC	Network-A		Network-B		Network-C	
	Online	Prep.	Online	Prep.	Online	Prep.
3PC	0.047	0.319	0.2	1.34	1.92	12.806
7PC	0.061	0.403	0.257	1.69	2.466	16.154
11PC	0.065	0.425	0.273	1.783	2.615	17.043
21PC	0.068	0.444	0.286	1.86	2.738	17.776
31PC	0.069	0.45	0.291	1.887	2.782	18.035
63PC	0.07	0.457	0.296	1.916	2.829	18.309

Table 4: Online and preprocessing communication size per party of oblivious inference. All numbers are reported in MB.

Protocol	Network-A		Network-B		Network-C	
	LAN	WAN	LAN	WAN	LAN	WAN
Falcon[45]	0.009	0.532	0.011	0.535	0.04	2.171
This	0.006	0.37	0.007	0.387	0.02	1.222
Times	1.5×	1.4×	1.6×	1.4×	1.9×	1.8×

Table 5: Comparing the performance of Falcon and our protocol.

Our protocol is specifically designed to support a large number of parties in PPML. The observed performance aligns with our expectations. Additionally, we compare our protocol with the state-of-the-art three-party protocol Falcon [45], which focuses on power-of-2-rings case. It indeed performs truncation as ABY3 [34], inheriting the large gap. For comparison protocol, the main idea of comparison is to indicate the first bit (from the MSB) where two numbers differ. Falcon uses a more indirect and complicated way to determine it with logarithmic rounds while we use our improved primitive (prefix-OR) with 1 round. Roughly speaking, the underlying reason is that some primitives are much more costly in the ring setting than in the field setting, e.g. prefix-multiplication.

In Table 5, we present a performance comparison between Falcon and our protocol for different networks in semi-honest honest-majority settings. It is worth mentioning that the source codes of Falcon [45] do not contain the preprocessing. In the online phase, our protocol demonstrates a speed improvement of 1.5× to 1.9× compared to Falcon in the LAN setting and 1.4× to 1.8× in the WAN setting.

Acknowledgments

Yu Yu was supported by the National Natural Science Foundation of China (Grant Nos.62125204, 92270201 and 61872236). This work has been supported by the New Cornerstone Science Foundation through the XPLORER PRIZE. Fengrun Liu would like to thank Junming Ma from the SPU team in Ant Group for his support.

References

[1] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 201–209, 1989.

[2] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology—CRYPTO'91: Proceedings 11*, pages 420–432. Springer, 1992.

[3] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 351–371. 2019.

[4] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. Flash: fast and robust framework for privacy-preserving machine learning. *Cryptology ePrint Archive*, 2019.

[5] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[6] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks: 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings 7*, pages 182–199. Springer, 2010.

[7] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511. IEEE, 2019.

[8] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: High throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 81–92, 2019.

[9] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. *arXiv preprint arXiv:1912.02631*, 2019.

[10] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spdz2k: Efficient mpc mod 2k for dishonest majority. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II*, pages 769–798. Springer, 2018.

[11] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2*, pages 342–362. Springer, 2005.

[12] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies*, 4:355–375, 2020.

[13] Anders PK Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium*, pages 2183–2200, 2021.

[14] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1102–1120. IEEE, 2019.

[15] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*, pages 285–304. Springer, 2006.

[16] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*, pages 1–18. Springer, 2013.

[17] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Advances in Cryptology-CRYPTO 2007: 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007. Proceedings 27*, pages 572–590. Springer, 2007.

[18] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference,*

Santa Barbara, CA, USA, August 19-23, 2012. Proceedings, pages 643–662. Springer, 2012.

[19] Ivan Damgård and Rune Thorbek. Non-interactive proofs for integer multiplication. In *Advances in Cryptology-EUROCRYPT 2007: 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007. Proceedings 26*, pages 412–429. Springer, 2007.

[20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II 40*, pages 823–852. Springer, 2020.

[21] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 307–328. 2019.

[22] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. Atlas: Efficient and scalable mpc in the honest majority setting. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II 41*, pages 244–274. Springer, 2021.

[23] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[24] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1651–1669, 2018.

[25] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[26] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.

[27] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part III*, pages 158–189. Springer, 2018.

[28] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. Swift: Super-fast and robust privacy-preserving machine learning. In *USENIX Security Symposium*, pages 2651–2668, 2021.

[29] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 336–353. IEEE, 2020.

[30] Yehuda Lindell and Ariel Nof. A framework for constructing fast mpc over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 259–276, 2017.

[31] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 619–631, 2017.

[32] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I*, pages 249–270. Springer, 2021.

[33] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: a cryptographic inference system for neural networks. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*, pages 27–30, 2020.

[34] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 35–52, 2018.

[35] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[36] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Public Key Cryptography–PKC 2007: 10th International Conference on Practice and Theory in Public-Key Cryptography Beijing, China, April 16-20, 2007. Proceedings 10*, pages 343–360. Springer, 2007.

[37] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority mpc by batchwise multiplication verification. In *Applied Cryptography and Network Security: 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings 16*, pages 321–339. Springer, 2018.

[38] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Aby2. 0: Improved mixed-protocol secure two-party computation. In *USENIX Security Symposium*, pages 2165–2182, 2021.

[39] Arpita Patra and Ajith Suresh. Blaze: blazing fast privacy-preserving machine learning. *arXiv preprint arXiv:2005.09042*, 2020.

[40] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 325–342, 2020.

[41] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E Lauter, and Farinaz Koushanfar. Xonn: Xnor-based oblivious deep neural network inference. In *USENIX Security Symposium*, pages 1501–1518, 2019.

[42] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia conference on computer and communications security*, pages 707–721, 2018.

[43] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[44] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019(3):26–49, 2019.

[45] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proceedings on Privacy Enhancing Technologies*, 2021(1):188–208, 2021.

[46] Christopher Burges Yann Lecun, Corinna Cortes. Mnist database. http://yann.lecun.com/exdb/mnist, 2017.

[47] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

A Protocol Hierachy

An overview of the different protocols considered in this work is informally described in Figure 1.

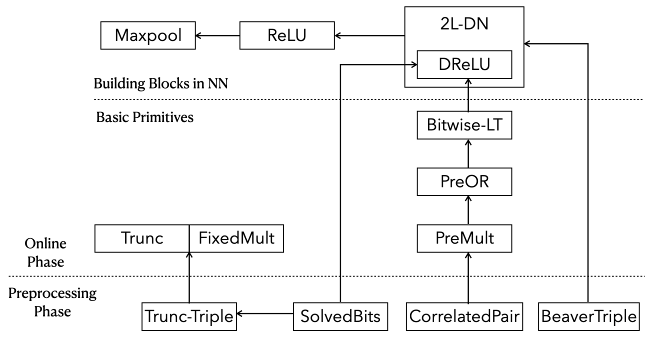

Figure 1: Hierachy among the protocols considered in this work. An arrow pointing from A to B means that protocol B requires protocol A.

Here we list the required randomness that can be generated in the preprocessing phase. These random materials are essentially derived from the basic random sharings and double random sharings in DN protocol (that are omitted here). For brevity, the notation r below denotes a uniformly random element in $\mathbb{F}_p$, so is r_i.

- Trunc-Triple: The truncation triple is in the form of $([r']_p, [r]_p, [r_{\text{msb}}]_p)$ or $([r']_p, \langle r \rangle_p, [r_{\text{msb}}]_p)$ where $r' = \text{Trunc}_d(r)$ and r_{msb} corresponds to the MSB of r.
- SolvedBits: This is a pair in the form of $([r]_p, [r]_{\text{B}})$.
- CorrelatedPair: The correlated pairs are a sequential of random sharings $\{([r_i]_p, [r'_i]_p)\}_{i=1}^t$ where $r'_1 = r_1^{-1}$ and $r'_i = r_{i-1} r_i^{-1}$ for $i > 1$ such that $r_i \Pi_{j=1}^i r'_i = 1$.
- BeaverTriple: The beaver triple is in the form of $([a]_p, [b]_p, [c]_p)$ such that $c = a \cdot b$.

B Functionality Description

Functionality B.1. $\mathcal{F}_{\text{Rand}}$

Input: The functionality receives no inputs.
Output: Compute the following

1. Randomly sample $r \in \mathbb{F}_p$.
2. Distribute shares of $[r]_p$ to the parties.

Functionality B.2. $\mathcal{F}_{\text{DoubleRand}}$

Input: The functionality receives no inputs.
Output: Compute the following

1. Randomly sample $r \in \mathbb{F}_p$.
2. Distribute shares of $[r]_p$ and $\langle r \rangle_p$ to the parties.

Functionality B.3. $\mathcal{F}_{\text{Mult}}$

Input: The functionality receives inputs $[a]_p$ and $[b]_p$.
Output: Compute the following

1. Reconstruct a and b to compute ab.
2. Distribute shares of $[ab]_p$ to the parties.

Functionality B.4. $\mathcal{F}_{\text{SolvedBits}}$

Input: The functionality receives no inputs.
Output: Compute the following

1. Randomly sample $r \in \mathbb{F}_p$.
2. Compute the decomposed bits $(r_0, r_1, \ldots, r_{\ell-1})$ such that $r = \sum_{i=0}^{\ell-1} r_i \cdot 2^i$.
3. Distribute the shares of $[r]_p$ and $[r]_{\text{B}}$ to the parties.

Functionality B.5. $\mathcal{F}_{\text{Trunc-Triple}}$

Input: The functionality receives no inputs.
Output: Compute the following

1. Sample a random element $r \in \mathbb{F}_p$ and compute $r' = \text{Trunc}_d(r)$ and r_{msb}.
2. Distribute the shares of this triple $([r']_p, [r]_p, [r_{\text{msb}}]_p)$ to the parties.

Functionality B.6. $\mathcal{F}_{\text{Trunc}}$

Input: The functionality receives inputs $[a]_p$.
Output: Compute the following

1. Reconstruct a.
2. Compute $b = \text{Trunc}_d(a)$.
3. Distribute shares of $[b]_p$ and send back to the parties.

Functionality B.7. $\mathcal{F}_{\text{Fixed-Mult}}$

Input: The functionality receives inputs $[a]_p$ and $[b]_p$.
Output: Compute the following

1. Reconstruct a and b.
2. Compute $c = ab$ and compute $d = \text{Trunc}_d(c)$.
3. Distribute shares of $[d]_p$ and send back to the parties.

Functionality B.8. $\mathcal{F}_{\text{PreMult}}$

Input: The functionality receives inputs $\{[a_i]_p\}_{i=1}^t$.
Output: Compute the following

1. Reconstruct $a_1, \ldots, a_t$.
2. Compute all prefix products $a_1, a_1 \cdot a_2, \ldots, \Pi_{i=1}^t a_i$.
3. Distribute the shares of these products to the parties.

Functionality B.9. $\mathcal{F}_{\text{MultPub}}$

Input: The functionality receives inputs $[a]_p$ and $[b]_p$.
Output: Compute the following

1. Reconstruct a and b to compute ab.
2. Send ab to the parties.

Functionality B.10. $\mathcal{F}_{\text{PreOR}}$

Input: The functionality receives inputs $[a_1]_p, \ldots [a_\ell]_p$ where $a_i \in \{0, 1\}$.
Output: Compute the following

1. Reconstruct $a_1, \ldots, a_\ell$.
2. Compute all prefix-`OR`s $a_1, a_1 \vee a_2, \ldots, \vee_{i=1}^{\ell} a_i$.
3. Distribute the shares of these prefix-`OR`s to the parties.

Functionality B.11. $\mathcal{F}_{\text{Bitwise-LT}}$

Input: The functionality receives inputs a and $[b]_{\text{B}}$ where $a, b \in \mathbb{F}_p$.
Output: Compute the following

1. Reconstruct $b_0, \ldots, b_{\ell-1}$ to compute $b = \sum_{i=0}^{\ell} 2^i \cdot b_i$.
2. Compute $c = (a < b)$ where $c \in \{0, 1\}$.
3. Distribute the shares of $[c]_p$ to the parties.

Functionality B.12. $\mathcal{F}_{\text{DReLU}}$

Input: The functionality receives inputs $[a]_p$.
Output: Compute the following

1. Reconstruct a to compute $b = \text{DReLU}(a)$.
2. Distribute the shares of $[b]_p$ to the parties.

C Security Proofs

We only describe the simulation for protocol $\Pi_{\text{Fixed-Mult}}$, and the simulation for protocol Π_{Trunc} is similar. The simulations for the remaining protocols are mostly simple compositions of local computations and invocations of ideal functionalities.

We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. In the beginning, $\mathcal{S}$ receives the input shares of $[a]_p$ and $[b]_p$ held by corrupted parties.

When invoking $\mathcal{F}_{\text{Trunc-Triple}}$, $\mathcal{S}$ invokes the simulator of $\mathcal{F}_{\text{Trunc-Triple}}$ and receives from the adversary the shares of $([r']_p, \langle r \rangle_p, [r_{\text{msb}}]_p)$ held by corrupted parties.

Functionality B.13. $\mathcal{F}_{\text{2L-DN}}$

Input: The functionality receives inputs $[x]_p, [y]_p$ and $\{[z^{(i)}]_p\}_{i=1}^m$.
Output: Compute the following

1. Reconstruct x, y and $\{z^{(i)}\}_{i=1}^m$ to compute $\{xyz^{(i)}\}_{i=1}^m$.
2. For $i = 1, \ldots, m$: Distribute the shares of $[xyz^{(i)}]_p$ to the parties.

Functionality B.14. $\mathcal{F}_{\text{ReLU}}$

Input: The functionality receives inputs $[a]_p$.
Output: Compute the following

1. Reconstruct a to compute $b = \text{ReLU}(a)$.
2. Distribute the shares of $[b]_p$ to the parties.

Functionality B.15. $\mathcal{F}_{\text{Maxpool}}$

Input: The functionality receives inputs $[a_1]_p, \ldots, [a_m]_p$.
Output: Compute the following

1. Reconstruct $a_1, \ldots, a_m$.
2. Compute $k = \text{argmax}\{a_1, \ldots, a_m\}$.
3. Distribute the shares of $[a_k]_p$ to the parties.

In Step 1, for each honest party, $\mathcal{S}$ samples a random element as its shares of $\langle ab + r + 2^{\ell-2} \rangle_p$. For each corrupted party, $\mathcal{S}$ computes its share of $\langle ab + r + 2^{\ell-2} \rangle_p$. In Step 2, depending on whether $P_{\texttt{pking}}$ is a corrupted party, there are two cases:

- If $P_{\texttt{pking}}$ is an honest party, $\mathcal{S}$ uses these shares to reconstruct the secret $c = ab + r + 2^{\ell-2}$, and sends c back to corrupted parties.

- If $P_{\texttt{pking}}$ is a corrupted party, $\mathcal{S}$ sends the shares of $[ab + r + 2^{\ell-2}]_p$ of honest parties to $P_{\texttt{pking}}$. $P_{\texttt{pking}}$ also receives the shares from corrupted parties. Then $P_{\texttt{pking}}$ can reconstruct c and send c back to corrupted parties.

In the following steps, $\mathcal{S}$ respectively computes the shares of $[e]_p$ and $[f]_p$ held by corrupted parties. Note that $\mathcal{S}$ has computed the shares of $[r_{\text{msb}}]_p$ and $[r']_p$ held by corrupted parties when simulating $\mathcal{F}_{\text{Trunc-Triple}}$.

In the real view, c is uniformly random since r is uniformly random. Hence, the view generated by the simulator $\mathcal{S}$ is indistinguishable from the real view underlying the security of $\mathcal{F}_{\text{Trunc-Triple}}$.

D Simulate Multi-party Computation on Limited Servers

We use 11 servers to simulate a range of parties. In the setting of 3PC, 7PC, and 11PC, a single party controls a single server. In the settings of 21PC, 31PC and 63PC, we run multi-parties on a single server simultaneously with different ports where we guarantee that the number of parties on different servers differs by up to one. We use the Linux `tc` command to set the latency and the bandwidth on both the Network Interface Controller (NIC) with regard to the IP address for parties on different servers and the local loopback NIC for parties on the same server, which ensures the ping-time and the throughput in each pair of parties is almost consistent.

E Neural Networks

Network-A Network-A is a 3-layer Deep Neural Network (DNN) from SecureML [35] which consists of 2 fully connected hidden layers with ReLU activations, each having 128 nodes along with a 10 node output layer.

Network-B Network-B is a 3-layer Convolutional Neural Network (CNN) derived from Chameleon [42] where the first layer is a 2-dimensional convolutional layer with a kernel of 5×5, stride of 2 (without padding), and 5 output channels. The activation function next is ReLU. The size of the kernel is optimized to 2×2 in Falcon [45]. The second hidden layer is a fully connected layer from a vector of size 980 to a vector of size 100, followed by ReLU activations. The last layer is a 10 node output layer from a vector of size 100 to a vector of size 10.

Network-C Network-C is a 4-layer CNN selected from MiniONN [31] with 2 convolutional and 2 fully-connected layers. The first layer is a 2-dimensional convolutional layer with 1 input channel, 16 output channels and a 5×5 kernel of stride of 1 (without padding). The activation functions following is a 2×2 Maxpool of stride 2, followed by ReLU activations, where the order of these two activation functions is swapped to optimize runtimes in Falcon [45]. The second layer is a 2-dimensional convolutional layer with 16 input channels, 16 output channels and another 5×5 kernel of stride 1 (without padding), followed by a 2×2 Maxpool of stride 2 and ReLU once again as activation functions. The third layer is a fully connected layer from a vector of size 256 to a vector of size 100 with ReLU activations. The last layer is a 10 node output layer as well.