Secure Floating-Point Training

Deevashwer Rathee[1], Anwesh Bhattacharya[2], Divya Gupta[2], Rahul Sharma[2], and Dawn Song[1]
[1]*University of California, Berkeley*
[2]*Microsoft Research*

Abstract

Secure 2-party computation (2PC) of floating-point arithmetic is improving in performance and recent work runs deep learning algorithms with it, while being as numerically precise as commonly used machine learning (ML) frameworks like PyTorch. We find that the existing 2PC libraries for floating-point support generic computations and lack specialized support for ML training. Hence, their latency and communication costs for compound operations (e.g., dot products) are high. We provide novel specialized 2PC protocols for compound operations and prove their precision using numerical analysis. Our implementation BEACON outperforms state-of-the-art libraries for 2PC of floating-point by over $6\times$.

1 Introduction

Deep Neural networks (DNNs) are now being deployed in domains with sensitive data, such as healthcare and finance. The more diverse data a DNN is trained on, the more useful it becomes. While diverse data can be obtained by pooling data of multiple organizations, it is challenging to do so due to privacy policies that restrict data sharing.

This motivates the problem of *secure training* that allows many mutually distrustful parties to collaboratively learn a DNN model of their secret data without revealing anything about their data. At the end of secure training, each party learns the model and nothing else[1] about the data of the other parties beyond what can be deduced from the model. The seminal work of SecureML [55] showed that secure training of DNNs can be solved by secure multi-party computation (MPC) and specifically by secure 2-party computation (2PC) in the *client-server* model [27, 56, 57] with two non-colluding servers (Section 3.3). In general, 2PC protocols [29, 73] allow two mutually distrustful parties to compute functions over their secret inputs with the guarantee that nothing beyond the function output is revealed about the sensitive inputs.

[1]Approaches based on trusted hardware and federated learning suffer from additional leakage and do not provide this cryptographic security guarantee.

Cleartext ML frameworks like PyTorch, when running on CPUs, *decompose* a training algorithm into many floating-point scalar operations or their SIMD[2] (Single Instruction Multiple Data) counterparts that are present in Intel's libraries. By running efficient and precise 2PC protocols for these scalar/SIMD floating-point operations, one can get secure training implementations that are as precise as PyTorch (see Section 4.2 for definition of precision) and have tractable latency, which is the approach taken by the recent work of SECFLOAT [60]. Note that SECFLOAT [60] is the current state-of-the-art in 2PC of floating-point that provably meets the precision specified by Intel's libraries and outperforms all prior 2PC libraries for secure floating-point (ABY [22], EMP [3], and MP-SPDZ [40]) by $3-230\times$. We observe that while running ML training algorithms with SECFLOAT, over 99% of the execution time is spent in linear layers, i.e., in convolutions and fully connected layers (Appendix A), which evaluate *compound operations* such as matrix multiplications or dot products[3]. Hence, to reduce the 2PC latency of ML training, we focus on building specialized secure protocols for these compound operations.

1.1 Our contributions

We provide novel specialized protocols for compound operations occurring in ML training that are as precise as SECFLOAT-based protocols while being much more efficient. Note that, apart from SECFLOAT, all other prior works in secure training use approximations that lack formal precision guarantees [16, 17, 41, 43, 54, 55, 64, 66, 67]. Among these, KS22 [41] is the state-of-the-art whose approximations have high efficiency and have been shown to empirically match the end-to-end training accuracy provided by floating-point training on MNIST [46]/CIFAR [44] datasets. We show that the latency overheads of our provably precise protocols are $< 6\times$ over KS22. In particular,

[2]For example, a SIMD addition of two vectors $(x_1,\ldots,x_n)$ and $(y_1,\ldots,y_n)$ gives $(x_1+y_1,\ldots,x_n+y_n)$.

[3]Also known as inner product.

- We provide the first specialized 2PC protocols for floating-point compound operations (e.g., dot products) and formally prove their accuracy using numerical error analysis.
- We implement our protocols in BEACON[4] and design them to be drop-in replacements for standard tensor operations (Section 8). BEACON is the first PPML library to support training over various floating-point representations, e.g., Google's BFloat16, Nvidia's TensorFloat FP19 and standard 32-bit floating-point FP32 (Appendix D).
- BEACON enables push button secure evaluation of PyTorch training algorithms. We evaluate BEACON on multiple models for MNIST and CIFAR-10 datasets. BEACON improves the latency of secure training by $> 6\times$ over the state-of-the-art libraries for 2PC of floating-point (Figure 5) and has $< 6\times$ overhead over KS22.

Note that BEACON enables secure n-party training in the standard client-server model with two non-colluding semi-honest servers [27, 55–57] (see Section 3.3). Researchers that wish to use BEACON for floating-point tasks outside the context of secure training should note that we have chosen to omit support for special values like subnormals [28] in BEACON as Intel's libraries with default compilation and prior 2PC floating-point implementations, including SECFLOAT, also don't support them [47, 60]. Now, we explain the main source of our efficiency gains at a high level.

1.1.1 Our protocols for precise compound operations

In addition to a sign-bit, a floating-point value consists of an integer and a fixed-point number, corresponding to the exponent and the mantissa, respectively. There are complex invariants which the exponent and the mantissa need to maintain to qualify as a valid floating-point value. During arithmetic operations, the intermediate results do not respect these invariants. Hence, to return a valid floating-point output, these invariants need to be restored through expensive rounding and normalization steps (Section 4). These steps are core to floating-point arithmetic, are necessary for precision, and are also the main reason behind performance overheads associated with 2PC of floating-point. In particular, for adding two floating-point numbers using SECFLOAT, over 82% of the time is spent in rounding and normalization (Appendix B). Now consider a compound operation, e.g., a summation on an $n+1$-length vector, i.e., given $x = (x_0, x_1, \ldots, x_n)$ compute $\sum_{i=0}^{n} x_i$. Decomposing a summation as n floating-point additions will require n rounding and n normalization steps. In contrast, we design our specialized protocols for compound operations to only require a *single* normalization and rounding operation while guaranteeing numerically precise results.

Every normalization that is not done is a threat to correctness and every rounding operation omitted increases the bitwidth of intermediate values, and in turn, the cost of subsequent operations. Reducing normalizations and rounding operations while guaranteeing precision and performance is challenging. We achieve our results by working over carefully determined minimal bitwidths needed for intermediate computations that balance precision and performance. We note that even though protocols for all compound operations such as summation and dot products are designed with the same goal of minimizing these expensive steps, they require different insights and numerical analyses to determine the exact parameters for efficiency and to prove precision guarantees.

1.1.2 BFloat16 training

The cleartext training algorithms are adopting low bitwidth floating-point representations such as the 16-bit BFloat16 or BF16 format. Compared to standard 32-bit floating-point representation, FP32, BF16 uses the same number of exponent bits as FP32, i.e., 8, but reduces the number of bits in mantissa from 23 to 7. In *BFloat16 training* [37], BF16 numbers are used to store activations and the arithmetic happens in FP32. For example, the specification of a matrix multiplication in BFloat16 training says that given two input matrices that are in BF16 format the output should be as precise as the result of the following computation: convert the inputs to FP32, then perform all the arithmetic in FP32, and then round the result to BF16. Kalamkar et al. [37] evaluate on many ML models and shows that BFloat16 training matches the accuracy of standard FP32 training. In fact, hardware manufacturers are putting native support for BFloat16 training in CPUs, GPUs, TPUs, etc. Note that the decomposition-based mechanism is incompatible with the compound operations in BFloat16 training. The compound operation is decomposed into scalar/SIMD operations over either FP32 or BF16. The former loses any performance advantage that secure BFloat16 training can provide over secure FP32 training and the latter loses precision as the operations on BF16 are over $65000\times$ less precise[5] than the operations over FP32, as required by the above specification.

In our protocols for BF16 compound operations, intermediate values use impure representations that are neither FP32 nor BF16. The bitwidths of these represenatations are carefully chosen to ensure that the computations are exact. These bitwidths are still lower than those required for FP32 computations and outperform BEACON's protocols for FP32. The impure representations do not satisfy the preconditions required by the protocols designed for pure representations. Hence, as another technical contribution we generalize our underlying protocols to handle impure representations.

Thus, BEACON provides two improvements over decom-

[4]Implementation available at https://github.com/mpc-msri/EzPC.

[5]A floating-point representation with q-bit matissa incurs a relative rounding error of 2^{-q-1} [28]. Since BF16 has 7 mantissa bits, it incurs a relative rounding error of 2^{-7-1}, while FP32 with 23-bit mantissa only incurs 2^{-23-1}, which is $65536\times$ lower.

posing BF16 compound operations to scalar FP32 operations and running them with SecFloat. For example, while summing 2000 BF16 values, we can either use Beacon's summation over FP32 or Beacon's specialized summation over BF16. The former is $8\times$ faster than SecFloat and the latter is $2\times$ faster than the former. Overall, our specialized protocol in Beacon for BF16 is $16\times$ faster than SecFloat for this task. Finally, our protocols for BFloat16 training are parameterized on the number representation and directly generalize to TensorFloat training, which is the same as BFloat16 training except that it uses Nvidia's 19-bit TensorFloat representation (8-bit exponent and 10-bit mantissa), FP19, instead of BF16.

1.2 Organization

The rest of the paper is organized as follows. Section 2 describes our protocols at a high level. Section 3 provides background on 2PC, our threat model, and 2PC protocols over integers. Section 4 provides background on floating-point and 2PC of scalar/SIMD operations over floating-point. Section 5 provides our protocols for compound operations over a unique number representation, e.g., operations that arise in FP32 training. Section 6 provides our protocols for compound operations that switch number representations, e.g., operations that arise in BFloat16 training. Section 7 argues security of Beacon, Section 8 discusses implementation details, and Section 9 evaluates Beacon on ML tasks. Section 10 discusses related work and Section 11 concludes with directions for future research.

2 Technical overview

In this section, we provide a high level intuition of our techniques. To ease exposition, we abstract out the actual floating-point representation (Section 4) that uses exponents and mantissas.

Novel protocol for summation. For secure training with SecFloat, linear layers (i.e., matrix multiplications and convolutions) are the performance bottlenecks and consume 99% of the runtime and communication (Appendix A). Here, over 85% (Appendix B) of matrix multiplication runtime is spent in summations (i.e., computing $\sum_{i=0}^{n} x_i$), which makes summation the main performance bottleneck of secure training.

As summarized in Section 1.1.1, traditional approaches for summation, including the ones used by PyTorch, require n rounding and n normalization steps that are expensive in 2PC and are the main performance bottlenecks. However, normalization and rounding are crucial for two reasons: (i) preventing overflows while operating on finite bits, and (ii) subsequent operations guarantee precision assuming a normalized floating-point input. Repeated rounding also leads to aggregation of rounding errors, and final relative error depends on n, the length of the summation performed. In particular, PyTorch can incur a relative error of $\varepsilon\kappa n$ where ε is *machine epsilon* [28] of the floating-point representation and κ is the *condition number* [65] of the summation problem (see Theorem 1 for exact definition), which is a real-valued quantity that is independent of how summation is implemented in finite-precision floating-point.

To address this performance bottleneck in summation, our first idea is to perform intermediate computations in large-enough bitwidths to replicate exact real arithmetic followed by one final normalization and rounding step. However, this approach requires the bitwidth to depend on the difference in magnitudes of the largest and the smallest values being added and could be as large as $276 + \log n$ bits for FP32. Hence, this turns out to be quite expensive in 2PC and also wasteful. In light of this, we set our goal to have smaller worst case relative error compared to traditional approaches, say, close to $\varepsilon\kappa$. With this error in mind, we carefully determine a threshold such that we can pick a reasonable bitwidth ℓ to ensure that (i) all values with magnitude larger than the threshold are exactly summed and (ii) ignoring all values with magnitude smaller than the threshold leads to at most $\varepsilon\kappa$ error. After doing summation of values that matter in ℓ bits, we perform one normalization and rounding. The final rounding leads to additional ε error and our overall relative error is at most $\varepsilon(\kappa+1)$. We note that unlike all traditional approaches, our error is independent of n. Our approach also results in $5\times$ fewer communication rounds over SecFloat (Section 5.1) for $n = 1000$.

BFloat16 training. Recall from Section 1.1.2 that in BFloat16 (or BF16) training, dot products (and matrix multiplications) multiply and sum BF16 values in FP32. However, simply performing all arithmetic in FP32 is wasteful and we miss out on any performance benefits of using BF16. Our starting point is the observation that the precision of the *exact product* of two BF16 values (14-bits) is smaller than the precision of FP32 (23-bits), and thus, we can compute over smaller intermediate bitwidths than in FP32. However, our FP32 summation protocol expects a precision of 23-bits. Consequently, it will pick a larger threshold leading to more values being ignored, and a higher (and unacceptable) final error. Hence, to provide the same guarantees as the underlying FP32 computation, we artificially increase the precision of intermediate products, which were computed *exactly*, to match that of FP32. We also remove the normalization step used in multiplications. Since this violates the input precondition of summation described above, we need to further generalize our summation protocol to accept unnormalized inputs without losing its benefits. Overall, we meet the specified precision and obtain a performance improvement of $1.7\times$ over dot product (length-1000 vectors) in FP32 (Table 7) that can be attributed to use of lower bitwidths in both the multiplications for intermediate products as well as the summation, and avoiding the use of operations like rounding.

Finally, even for scalar/SIMD non-linear activations such as Sigmoid and Tanh, the specification requires computations over FP32 followed by rounding to BF16. We exploit the lower bitwidth of BF16 and domain knowledge to provide efficient protocols that beat the approach of computing intermediate results over FP32 by $5\times$.

3 Preliminaries

We define notation, secret sharing, threat model for secure training followed by 2PC building blocks over integers.

3.1 Notation

We denote the computational security parameter by λ and $[n]$ denotes the set $\{1,\dots,n\}$. Variable names with Roman letters (a, b, etc.) are integer-typed and those with Greek letters (α, β, etc.) are floating-point. The indicator function $\mathbf{1}\{P\}$ returns 1 if the predicate P is true and 0 otherwise; $x||y$ concatenates bit-strings x and y. An ℓ-bit integer $x \in \mathbb{Z}_{2^\ell}$ can be interpreted as an unsigned integer $\mathsf{uint}(x) = \zeta_\ell(x)$, where ζ_ℓ is a lossless lifting operator from bitstrings in $\mathbb{Z}_{2^\ell}$ to integers in $\mathbb{Z}$. Using the 2's complement encoding, x can also be interpreted as a signed integer $\mathsf{int}(x) = \mathsf{uint}(x) - \mathsf{MSB}(x)\cdot 2^\ell$, where $\mathsf{MSB}(x)$ is the most-significant bit of x.
Fixed-point: An unsigned fixed-point number $x \in \mathbb{Z}_{2^\ell}$ with bitwidth ℓ and scale s represents the real number $\frac{\mathsf{uint}(x)}{2^s} \in \mathbb{Q}$.

3.2 Secret Sharing

Secret sharing [9, 63] is a technique of distributing a secret among a group of parties by allocating each party a share of the secret. In this work, we consider 2-out-of-2 additive secret sharing where a secret $x \in \mathbb{Z}_{2^\ell}$ is split into two shares $\langle x\rangle^\ell = (\langle x\rangle_0^\ell, \langle x\rangle_1^\ell) \in (\mathbb{Z}_{2^\ell})^2$ and party P_i holds $\langle x\rangle_i^\ell$. Each secret share $\langle x\rangle_i^\ell$ has the property that it reveals nothing about the secret x in isolation but the shares can all be put together to reconstruct the secret x as follows: $x = \langle x\rangle_0^\ell + \langle x\rangle_1^\ell \bmod 2^\ell$. We use the superscript B to denote secret-shares of boolean values $\in \mathbb{Z}_2$.

3.3 2PC and Threat Model

Secure 2-party computation (2PC) introduced by [29, 73] allows two parties P_0 and P_1 to compute an agreed upon function f on their sensitive inputs x and y, respectively. It provides an interactive protocol with the strong guarantee that the interaction reveals nothing about the sensitive inputs beyond what can be inferred from the output. A common approach for 2PC is where parties begin by secret sharing their inputs with the other party and run a protocol that takes shares of (x,y) to securely generate shares of $f(x,y)$. Then, the parties exchange the shares of the output and reconstruct the output. With this design in mind, it is sufficient to construct 2PC protocols, for operations in machine learning, that go from shares of input to shares of output securely.
2PC threat model. Our threat model is same as SecureML [55] and considers a static probabilistic polynomial-time (PPT) semi-honest adversary. It corrupts one of the parties (P_0 or P_1) at the beginning of the protocol and tries to learn information about honest party's sensitive input while following the protocol specification faithfully. We argue security against this adversary in the standard simulation paradigm [13, 29, 48] that argues indistinguishability of the views of the adversary in the real execution and ideal execution in the presence of a trusted third party that takes inputs and provides the function outputs alone.
Client-Server Model for secure training of DNNs [55] considers n clients with sensitive training data (C_i has x_i, $i \in [n]$) and 2 inputless servers, S_0 and S_1. The goal is for the clients to learn the output of an ML training algorithm $y = f(x_1,\dots,x_n)$. We consider an static semi-honest adversary that corrupts at most one server and $n-1$ clients. For secure training in this setting, the clients first secret share their inputs among two servers S_0 and S_1, who run the 2PC protocol for training to obtain shares of y that are sent to the clients. Clients reconstruct y to learn the output of the training algorithm.

3.4 Integer Building Blocks

Table 1 describes the fixed-point or integer operations that BEACON uses in its protocols for floating-point compound operations. The communication and round costs[6] of the corresponding protocols are given in Table 2. Improvement in these costs will directly improve BEACON. All of these operations except Round-Nearest (RN) have been considered and discussed in detail in the cited works. RN rounds an ℓ-bit value to a $(\ell-s)$-bit value. Note that SECFLOAT [60] provided a more complex protocol that achieves round to nearest and ties to even, which was required for its correctness guarantees. For our precision guarantees (Section 4.2), a simpler function $\mathsf{RN}(x,s) = \lfloor \frac{x}{2^s} \rceil$, that does round to nearest and ties are always rounded up suffices and is also cheaper in 2PC. It is easy to see that $\lfloor \frac{x}{2^s} \rceil = \lfloor \frac{x+2^{s-1}}{2^s} \rfloor$, and hence, we implement our protocol as $\Pi_{\mathsf{RN}}^{\ell,s}(\langle x\rangle^\ell) = \Pi_{\mathsf{TR}}^{\ell,s}(\langle x\rangle^\ell + 2^{s-1})$, where $\Pi_{\mathsf{TR}}^{\ell,s}$ is the truncate-reduce protocol from [61] that rounds down by truncating the lower s bits of x.

4 Floating-point Background

We review the basics of floating-point representation and the precision guarantees of floating-point implementations. Then we define the secret sharing of floating-point values and the helper protocols we use.

[6]The MSB-to-Wrap optimization from SIRNN [61] is applicable to all instances of extension and multiplication, and thus, we report the optimized costs for these building blocks.

Functionality	Notation		Description
	Algorithm	Protocol	
Integer/Fixed-point Building Blocks			
Multiplexer [62]	$z = c\ ?\ x : y$	$\langle z\rangle^\ell = \Pi^\ell_{\mathsf{MUX}}(\langle c\rangle^B, \langle x\rangle^\ell, \langle y\rangle^\ell)$	$z = x$ if $c = 1$, else $z = y$
Less-Than [62]	$c = \mathbf{1}\{x < y\}$	$\langle c\rangle^B = \Pi^\ell_{\mathsf{LT}}(\langle x\rangle^\ell, \langle y\rangle^\ell)$	Checks if $x < y$, $x, y \in \mathbb{Z}_{2^\ell}$
LT&EQ [60]	$c = \mathbf{1}\{x < y\}$ $e = \mathbf{1}\{x = y\}$	$\langle c\rangle^B, \langle e\rangle^B = \Pi^\ell_{\mathsf{LT\&EQ}}(\langle x\rangle^\ell, \langle y\rangle^\ell)$	Checks if $x < y$ and if $x = y$, $x, y \in \mathbb{Z}_{2^\ell}$
Zero-Extension [61]	$y = \mathsf{ZXt}(x, n)$	$\langle y\rangle^n = \Pi^{m,n}_{\mathsf{ZXt}}(\langle x\rangle^m)$	$\zeta_n(y) = \zeta_m(x) \bmod 2^n, m \leqslant n$
Round-Nearest (Section 3.4)	$y = \mathsf{RN}(x, s)$	$\langle y\rangle^{\ell-s} = \Pi^{\ell,s}_{\mathsf{RN}}(\langle x\rangle^\ell)$	Upper $\ell - s$ bits of $(x + 2^{s-1})$
Most Significant Non-Zero Bit [61, 72]	$k, K = \mathsf{MSNZB}(x)$	$\langle k\rangle^\ell, \langle K\rangle^\ell = \Pi^\ell_{\mathsf{MSNZB}}(\langle x\rangle^\ell)$	k, s.t. $x_k = 1 \wedge \forall i > k, x_i = 0$, $K = 2^{\ell-1-k}$
Lookup Table (LUT) [23]	$y = L(x), y \in \mathbb{Z}_{2^n}$	$\langle y\rangle^n = \Pi^{m,n}_{\mathsf{LUT}}(L, \langle x\rangle^m)$	index x, LUT L, $z \in \mathbb{Z}_{2^n}$
Unsigned Mixed-bitwidth Multiplication [61]	$z = x *_\ell y$	$\langle z\rangle^\ell = \Pi^{m,n,\ell}_{\mathsf{UMult}}(\langle x\rangle^m, \langle y\rangle^n)$	$\zeta_\ell(z) = \zeta_m(x) \cdot \zeta_n(y) \bmod 2^\ell$, $\ell \geqslant \mathsf{max}(m, n)$
Find Maximum [60]	$y = \mathsf{max}_{i\in[n]} x_i$	$\langle y\rangle^\ell = \Pi^{\ell,n}_{\mathsf{max}}(\{\langle x_i\rangle^\ell\}_{i\in[n]})$	Find the maximum x_i
Floating-point Building Blocks			
Normalize [60]	$m', e' = \mathsf{Normalize}^{p,q,Q}(m, e)$	$\langle m'\rangle^{Q+1}, \langle e'\rangle^{p+2} = \Pi^{p,q,Q}_{\mathsf{Normalize}}(\langle m\rangle^{Q+1}, \langle e\rangle^{p+2})$	Section 4.4.1
Round&Check [60]	$m', e' = \mathsf{Round\&Check}^{p,q,Q}(m, e)$	$\langle m'\rangle^{q+1}, \langle e'\rangle^{p+2} = \Pi^{p,q,Q}_{\mathsf{Round\&Check}}(\langle m\rangle^{Q+1}, \langle e\rangle^{p+2})$	Section 4.4.2
Clip [60]	$\alpha = \mathsf{Clip}^{p,q}(z, s, e, m)$	$\langle \alpha\rangle^{\mathsf{FP}(p,q)} = \Pi^{p,q}_{\mathsf{Clip}}(\langle z\rangle^B, \langle s\rangle^B, \langle e\rangle^{p+2}, \langle m\rangle^{q+1})$	Section 4.4.3

Table 1: 2PC building blocks used by BEACON.

Protocol	Communication	Rounds
Π^ℓ_{MUX} [62]	$2\lambda + 2\ell$	2
Π^ℓ_{LT} [62]	$\lambda\ell + 14\ell$	$\log(\ell)$
$\Pi^\ell_{\mathsf{LT\&EQ}}$ [60]	$\lambda(\ell + 3) + 14\ell + 60$	$\log(\ell) + 2$
$\Pi^{m,n}_{\mathsf{ZXt}}$ [61]	$2\lambda - m + n + 2$	4
$\Pi^{\ell,s}_{\mathsf{RN}}$ (Section 3.4)	$\lambda(s + 1) + \ell + 13s$	$\log(s) + 2$
$\Pi^\ell_{\mathsf{MSNZB}}$ [61, 72]	$\lambda(5\ell - 4) + \ell^2$	2
$\Pi^{m,n}_{\mathsf{LUT}}$ [23]	$2\lambda + 2^m n$	2
$\Pi^{m,n,\ell}_{\mathsf{UMult}}$ [61]	$\lambda(2\mu + 6) + \mu(\mu + 2\nu)$ $+3\mu + 2\nu + 4$	4
$\Pi^{\ell,n}_{\mathsf{max}}$ [60]	$(n-1)\cdot(\Pi^\ell_{\mathsf{LT}} + \Pi^\ell_{\mathsf{MUX}})$	$\log(n)\cdot(\Pi^\ell_{\mathsf{LT}} + \Pi^\ell_{\mathsf{MUX}})$
$\Pi^{p,q,Q}_{\mathsf{Normalize}}$ [60]	$\lambda(7Q + 3) + (Q + 1)^2$	4
$\Pi^{p,q,Q}_{\mathsf{Round\&Check}}$ [60]	$\lambda(2Q - q + 6) + 28Q$ $-11q + 2p + 21$	$\log(Q + 1) + 4$
$\Pi^{p,q}_{\mathsf{Clip}}$ [60]	$\lambda(p + 6) + 2q + 16p + 34$	$\log(p + 2) + 2$

Table 2: Cost of 2PC building blocks used by BEACON. All communication is in bits. $\mu = \mathsf{min}(m, n), \nu = \mathsf{max}(m, n)$.

4.1 Floating-Point Representation

According to the IEEE-754 standard [4], a floating-point value $\alpha = (-1)^s \cdot 2^{e-\mathsf{bias}} \cdot (1 + \frac{m}{2^q})$ is represented as $s||e||m \in \{0,1\}^{p+q+1}$, where s is the sign-bit, e is the (biased) p-bit exponent, bias $= 2^{p-1} - 1$, and m is the q-bit mantissa. There are several floating-point representations defined by tuples (p, q), which define the range and precision, respectively. For instance, the representation tuple for FP32 is $(8, 23)$, for BF16 is $(8, 7)$, and for FP19 is $(8, 10)$. Rounding to (p, q) representation introduces a relative error of at most $\varepsilon = 2^{-q-1}$, that is referred to as the *machine epsilon*.

4.2 Precision of floating-point operations

The IEEE-754 standard precisely defines the output of FP32 scalar operations, and the handling of error conditions (underflows, overflows, etc.) using special values (infinities, NaNs, and subnormals) and operations on these special values (e.g., $\infty - \infty$ is NaN). However, the IEEE standard makes no comments about the compound operations, which is the subject of our study here.

We make two remarks. First, in numerical analysis textbooks [65], the precision of compound operations is established with pen-and-paper proofs that bound the relative error between the output of a floating-point implementation and the ideal Real result. Second, in ML training implementations inside PyTorch, the floating-point implementations lack the handlers for special values. Hence, to prove the precision of our protocols, we prove bounds on the relative error between the output α computed by the protocol and the ideal Real result r, assuming that special values do not arise during the computations. Recall that relative error between α and r is $|\frac{\alpha - r}{r}|$. We say that a protocol for a compound operation is *precise* if the worst case relative error of the protocol output is the same or lower than the worst case relative error of the output produced by the standard textbook implementations of the operator. Note that all floating-point implementations incur numerical errors, including the implementations in PyTorch and BEACON, and performing several operations in sequence accumulates the worst case errors. Given two implementations, the one with the lower (or, same) worst case relative error is said to be *more precise*.

4.3 Secret sharing of floating-point values

Identically to [60], we represent a floating-point value α parameterized by $p, q \in \mathbb{Z}^+$ as a 4-tuple $(\alpha.z, \alpha.s, \alpha.e, \alpha.m)$ where $\alpha.z = \mathbf{1}\{\alpha = 0\}$ is the zero-bit, $\alpha.s \in \{0,1\}$ is the sign-bit (set if $\alpha \leqslant 0$), $\alpha.e \in \{0,1\}^{p+2}$ is the (unbiased) exponent[7] taking values in $[-2^{p-1}+1, 2^{p-1})$, and $\alpha.m \in \{0,1\}^{q+1}$ is the normalized fixed-point mantissa with scale q taking values from $[2^q, 2^{q+1}-1] \cup \{0\}$. Note that while IEEE-754 standard stores just q bits of mantissa m' as the leading bit is always 1 (implicitly), we instead explicitly store the mantissa in $q+1$ bits along with the leading bit, i.e, $\alpha.m = 2^q + m'$. Consistent with this notation, a secret shared floating-point value α is a tuple of shares $\langle\alpha\rangle^{\mathsf{FP}(p,q)} = (\langle\alpha.z\rangle^B, \langle\alpha.s\rangle^B, \langle\alpha.e\rangle^{p+2}, \langle\alpha.m\rangle^{q+1})$. Finally, the Real value of α, i.e., $[\![\alpha]\!]$ is zero if $\alpha.z = 1$ and $(-1)^{\alpha.s} \cdot 2^{\mathsf{int}(\alpha.e)} \cdot \frac{\mathsf{uint}(\alpha.m)}{2^q}$ otherwise.

4.4 Floating-point Building Blocks

Here we discuss sub-operations that are used by scalar operations over floating-point in [60] and our compound operations. 2PC protocols for these can be built using the integer building blocks in Table 1 (see Appendix G).

4.4.1 Normalize

A mantissa is said to be normalized if its most significant bit (MSB) is 1. During floating-point operations, the mantissa can become *unnormalized*, i.e., its bit-representation will have z zeros in the most-significant-bits. A normalization step adjusts the exponent by decrementing it by z and left shifts the mantissa by z to get rid of the leading zeros. Note that computing z requires computing the location of the most significant non-zero bit (MSNZB) of the mantissa, which is an expensive operation in 2PC (Table 1). The normalization protocol $\Pi^{p,q,Q}_{\mathsf{Normalize}}$ (Appendix G.1) takes as input a $p+2$-bit exponent and a $Q+1$-bit mantissa with scale q. It returns a normalized mantissa over $Q+1$ bits with scale Q and the adjusted exponent.

4.4.2 Round&Check

We need to round a normalized mantissa $m \in [2^Q, 2^{Q+1})$ in higher precision Q to a normalized mantissa $m' \in [2^q, 2^{q+1})$ in lower precision q, while incurring a relative error $\leqslant \varepsilon = 2^{-q-1}$. However, simply using $\Pi^{Q+1,Q-q}_{\mathsf{RN}}$ can result in an unnormalized mantissa ($= 2^{q+1}$) and overflow $q+1$ bits if m is very close to 2^{Q+1} (see Section V-B in [60] for details). Thus, we use Round&Check protocol $\Pi^{p,q,Q}_{\mathsf{Round\&Check}}$ that additionally performs this check and adjusts the exponent accordingly (Appendix G.2).

[7]The exponent is stored in $p+2$-bits like [60] to ensure that all exponent related comparisons can be performed without overflowing the modulus.

4.4.3 Clip

Clipping is used to set results that have a smaller magnitude than the smallest representable value to 0. For instance, FP32 can only represent normal values in the range $[2^{-126}, 2^{128})$. Thus, our clipping protocol $\Pi^{p,q}_{\mathsf{Clip}}$ sets inputs with exponent < -126 to 0 (Appendix G.3). Note that handling subnormal numbers, i.e., those with magnitude less than 2^{-126} is straightforward (subnormals are fixed-point numbers with scale 126), but it leads to additional performance overheads and is neither supported by BEACON nor by the SECFLOAT baseline. Some GPUs don't support subnormals at all and others provide the "fast mode" that, like BEACON, clips subnormals to zero [70].

5 Compound Operations in ML

We describe our protocols for secure compound operations. We begin by discussing our novel protocol for Summation that sums up the values in a vector and is the backbone for all linear layers. While the techniques from this section work for general (p,q), it might be useful for a reader to keep in mind the example of FP32, i.e., $p = 8, q = 23$. We defer the discussion of non-linear operations to Appendix H (ReLU, Softmax, etc.) and focus on linear layers where 99% of training time is spent. Later, in Section 6, we will discuss the techniques and further optimizations specific to BFloat16.

5.1 Summation

Given a floating-point vector $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)$, where each α_i is a floating-point number, Summation computes $\gamma = \sum_{i=1}^n \alpha_i$. Before we discuss 2PC protocols for Summation for vectors of length $n > 2$, we first recall the (high-level) steps involved for the case of $n = 2$, i.e., the addition of 2 floating-point values parameterized by (p,q):

1. Compare the exponents of the operands and compute their difference, say d.
2. Left-shift the mantissa of the operand with larger exponent by d. This step aligns both the mantissas with the corresponding exponent as the smaller exponent.
3. Add/subtract the aligned mantissas based on the sign bits.
4. Use Normalize algorithm (Section 4.4) to normalize the mantissa to lie in $[1,2)$ and adjust the exponent (if needed).
5. Round the normalized mantissa, which is in higher precision, to the required precision q (introducing error in computation) using Round&Check algorithm (Section 4.4).
6. *Clip* the values smaller than the smallest representable floating-point number to 0 with Clip (Section 4.4).

Given the floating-point addition algorithm, the most suitable[8] option to compute floating-point Summation is

[8]Kahan summation [36] incurs worst-case error of $2\varepsilon\kappa$ but requires $O(n)$ normalization and rounding steps, making it more expensive.

```
Algorithm FPSum^{p,q,n}({α_i}_{i∈[n]})
1:  $\tilde{n} = \log(n)$; $\ell = 2q + 2\tilde{n} + 3$
2:  $e_{\mathsf{max}} = \mathsf{max}_i\ \alpha_i.e$
3:  $e_{\mathsf{thr}} = e_{\mathsf{max}} - q - \tilde{n} - 1$
4:  for $i \in [n]$ do
5:      $m_i = (\alpha_i.e < e_{\mathsf{thr}})\ ?\ 0 : \alpha_i.m$
6:      $m_i^{(\mathsf{align})} = m_i \ll (\alpha_i.e - e_{\mathsf{thr}})$
7:      $m_i^{(s)} = \alpha_i.s\ ?\ -m_i^{(\mathsf{align})} : m_i^{(\mathsf{align})}$
8:  $M^{(s)} = \sum_{i\in[n]} m_i^{(s)}$
9:  $(s, z) = \mathsf{LT\&EQ}^{\ell}(M^{(s)}, 0)$
10: $M = s\ ?\ -M^{(s)} : M^{(s)}$
11: $(M_1, e_1) = \mathsf{Normalize}^{p,q,\ell-1}(M, e_{\mathsf{thr}})$
12: $(M_2, e_2) = \mathsf{Round\&Check}^{p,q,\ell-1}(M_1, e_1)$
13: Return $\mathsf{Clip}^{p,q}(z, s, e_2, M_2)$
```

Figure 1: *Floating-Point Summation.*

tree-sum (or pairwise summation), that recursively computes $\gamma_n = \gamma_{\frac{n}{2},1} + \gamma_{\frac{n}{2},2}$, where $\gamma_{\frac{n}{2},1} = \sum_{i=1}^{\frac{n}{2}} \alpha_i$ and $\gamma_{\frac{n}{2},2} = \sum_{i=1}^{\frac{n}{2}} \alpha_{\frac{n}{2}+i}$. However, keeping in mind the above blueprint, even tree-sum has three drawbacks: (i) number of cryptographically expensive operations like normalization (step 4) and rounding (step 5), as well as clip (step 6), scale linearly with n, (ii) the worst-case relative error compared to the computation over reals is proportional to $\log(n)$, and (iii) the number of communication rounds are $\log(n)$ times the round complexity for a single floating-point addition.

Our Algorithm. We propose an algorithm for floating-point Summation that addresses all three drawbacks. The key insight in our algorithm is that the expensive steps, i.e., normalization, rounding and clipping, can be performed only once for the whole vector, as opposed to once per addition. This is because we only require the final output to be a normalized floating-point value and no such guarantees are required for intermediate values. This not only reduces the cryptographic cost greatly (by avoiding unnecessary normalization, rounding and clips), but, as we formally prove, also makes the worst-case error independent of n. Finally, the round complexity of the resulting 2PC protocol is $\log(n)$ times the rounds for comparison and multiplexer on $p+2$ bits, plus a constant. Note that this is much lower than the tree-sum discussed above. In the remainder of this section, we first describe our algorithm for Summation, then prove its worst case error bounds, and finally describe a 2PC protocol for Summation over secret shared floating-point.

Our Summation algorithm $\mathsf{FPSum}^{p,q,n}$ is described formally in Figure 1 and has the following steps:

1. Compare the exponents of all vector elements to find the largest exponent e_{max} (Step 2). For $\tilde{n} = \log(n)$, define $e_{\mathsf{thr}} = e_{\mathsf{max}} - q - \tilde{n} - 1$ and threshold $\Gamma = 2^{e_{\mathsf{thr}}}$ such that only vector elements with magnitude $\geqslant \Gamma$ contribute to the sum (Step 3). Thus, we set the mantissa of all values with exponent $< e_{\mathsf{thr}}$ to 0 (Step 5).
2. For all i, left-shift the mantissa of α_i by $(\alpha_i.e - e_{\mathsf{thr}})$, essentially setting the exponents of all elements to e_{thr} (Step 6). Note that a bitwidth of $2q + \tilde{n} + 2$ suffices for all left-shifted mantissas as $(\alpha_i.e - e_{\mathsf{thr}}) \leqslant q + \tilde{n} + 1$.
3. To be able to simply add the mantissas now, we convert the unsigned mantissas $m_i^{(\mathsf{align})}$ to signed mantissas $m_i^{(s)}$ by multiplexing using the sign bit $\alpha_i.s$ (Step 7).
4. Next, simply add the n aligned *signed* mantissas (Step 8). This step requires additional $\tilde{n}+1$ bits, i.e., needs to be done in $\ell = 2q + 2\tilde{n} + 3$ bits to ensure no overflows.
5. Find the sign of the resulting signed mantissa $M^{(s)}$ and whether it is equal to 0 by a (signed) comparison with 0 and set the sign bit s and zero bit z accordingly (Step 9).
6. Use the sign bit to revert the mantissa $M^{(s)}$ to unsigned value (Step 10).
7. Use Normalize algorithm (Section 4.4) to normalize the ℓ-bit output mantissa (with scale q) to lie in $[1, 2)$ with scale $(\ell - 1)$ and adjust the exponent accordingly (Step 11).
8. Round the normalized mantissa which is in higher precision $(\ell - 1)$ to the required precision q (Step 12) using Round&Check algorithm (Section 4.4).
9. Finally, clip the values smaller than the smallest representable floating-point number with (p, q) to 0 (Step 13) using Clip algorithm (Section 4.4).

The value of e_{thr} has been carefully chosen to help obtain low numerical error in Theorem 1. This algorithm assumes a relation between p and n. In particular, e_{thr} needs to fit in $(p+1)$ bits. Concretely, for $p = 8$ (that is true for FP32, BF16, FP19), n needs to be smaller than 2^{105} that trivially holds for any practical summation.

Observe that the algorithm invokes the expensive steps of normalize, round and clip only once, instead of n times in the tree sum algorithm. Next, we report the precision of our algorithm and a description of the corresponding 2PC protocol with its complexity.

Theorem 1. *The relative error of Figure 1 is at most* $\varepsilon \cdot (\kappa + 1) + O(\varepsilon^2 \kappa)$*, where* $\varepsilon = 2^{-q-1}$ *is machine epsilon and* $\kappa = \frac{\sum_{i\in[n]} |\alpha_i|}{\sum_{i\in[n]} \alpha_i}$ *is the condition number of summation.*

Proof. Let $\hat{\gamma}$ and γ represent the output of FPSum and result of summation over reals, respectively. Let α_k be the element with the largest exponent. Note that $|\alpha_k| \geqslant 2^{e_{\mathsf{max}}}$ and up until the rounding step (before Step 12), FPSum computes the sum γ' of all elements $\geqslant 2^{e_{\mathsf{thr}}}$ exactly. Thus, the total magnitude of elements set to 0 in Step 5 that are ignored by FPSum is at most $n \cdot 2^{e_{\mathsf{thr}}} \leqslant 2^{e_{\mathsf{max}}} \cdot 2^{-q-1} \leqslant |\alpha_k| \cdot 2^{-q-1}$. Hence, $|\gamma - \gamma'| \leqslant |\alpha_k| \cdot \varepsilon$, where $\varepsilon = 2^{-q-1}$ is the machine epsilon, and the relative error of γ' is $|\Delta| = \frac{|\gamma-\gamma'|}{|\gamma|} \leqslant \frac{|\alpha_k|\cdot\varepsilon}{\sum_{i\in[n]}\alpha_i} \leqslant \frac{\sum_{i\in[n]}|\alpha_i|\cdot\varepsilon}{\sum_{i\in[n]}\alpha_i} = \varepsilon \cdot \kappa$.

The final output $\hat{\gamma}$ of FPSum is obtained after rounding γ' to q mantissa bits. Thus, $\hat{\gamma} = \gamma'(1+\delta)$ [28], where $|\delta| \leqslant \varepsilon$, and the absolute error of $\hat{\gamma}$ w.r.t. γ is:

$$\begin{aligned}|\gamma - \hat{\gamma}| &= |\gamma - \gamma'(1+\delta)| = |\gamma - \gamma(1+\Delta)(1+\delta)| \\ &= |\gamma(\delta + \Delta + \delta\Delta)| \leqslant |\gamma| \cdot (\varepsilon + \varepsilon\kappa + \varepsilon^2\kappa).\end{aligned}$$

Thus, the relative error of $\hat{\gamma}$ is $\frac{|\gamma-\hat{\gamma}|}{|\gamma|} \leqslant \varepsilon(\kappa+1) + O(\varepsilon^2\kappa)$. $\square$

Since our algorithm FPSum incurs lower worst-case error than the linear summation, there is a potential optimization opportunity that explores trading off precision for better performance. For instance, to get the same worst-case error bound as linear summation, we can set $e_{\mathsf{thr}} = e_{\mathsf{max}} - q - 2$ (instead of $e_{\mathsf{max}} - q - \tilde{n} - 1$) and $\ell = 2q + \tilde{n} + 4$ (instead of $2q + 2\tilde{n} + 3$). This leads in reduction of bitwidth by $\tilde{n} - 1$ in various steps, which could lead to a significant improvement if n is large (e.g. 14% if $n = 2^{10}$). We do not use this optimization and use FPSum as is.

Secure Protocol. Figure 2 describes the 2PC protocol corresponding to our Summation algorithm FPSum (Figure 1). For each step of the algorithm in Figure 1, we invoke the corresponding 2PC protocol that takes the shares of the inputs and returns the shares of the outputs (see Section 3.4 for the 2PC building blocks used for this transformation). With this, the transformation from the algorithm to the 2PC protocol is straightforward for all steps except the left-shift step needed to align the mantissas (Step 6, Figure 1) and additional extension needed before adding the signed mantissas in Step 8, Figure 1. In Figure 2, steps 7-9 implement the left-shift of mantissas. Note that left-shift by r-bits can be implemented by multiplying the value by 2^r (in sufficient bitwidth). Also, since the shift amount for a mantissa is secret-shared, we need to compute the power-of-2 operation for a secret-shared input. Since we have a bound on the shift amount, we can implement it efficiently using a lookup table. In more detail, our algorithm left-shifts the i^{th} mantissa m_i by $r_i = (\alpha_i.e - e_{\mathsf{thr}}) < (q + \tilde{n} + 2)$. We store r_i in $k = \lceil \log(q + \tilde{n} + 2) \rceil$ bits (using a modulo by 2^k in Step 7), which we assume is smaller than $p + 2$, as is the case for all floating-point representations used in practice. Consider a lookup table pow2 with 2^k entries over $\{0,1\}^{q+\tilde{n}+2}$ such that $\mathsf{pow2}[i] = 2^i$. We do a secret lookup in pow2 at index r_i to learn shares of 2^{r_i}. Then, we multiply with m_i to obtain $m_i^{(\mathsf{align})}$ in $2q + \tilde{n} + 2$ bits. Next, to avoid overflows during addition of (aligned) mantissas and to accommodate the sign-bit, we require additional $\tilde{n} + 1$ bits. Hence, in Step 10, we extend the mantissa to $\ell = 2q + 2\tilde{n} + 3$ bits. This protocol computes the same result as FPSum and we state security in Section 7.

Complexity. As can be observed, the round complexity of our protocol is equal to the round complexity of $\Pi_{\mathsf{max}}^{p+2,n}$ (to compute the maximum exponent) plus *roughly* the round complexity of a single floating point addition. In contrast, the round complexity of tree-sum based protocol is roughly $\log n$ times the round complexity of a floating-point addition. Concretely, for FP32, we require $6\log n + 73$ rounds compared to $69 \log n$ in SECFLOAT that uses tree sum. Moreover, communication complexity of our protocol for FP32 and $n = 2000$ is 4.4× lower than SECFLOAT (Figure 5).

5.2 Generalized Summation

Our Summation algorithm in Section 5.1 requires that the inputs are normalized floating-point values with the same (p,q) parameters as the desired output. We propose a *generalized* Summation that can sum up unnormalized floating-point values. As we will see later, this generalized Summation would play a crucial role in our protocols for dot products and BFloat16 datatype as well.

First, we set up the problem statement. Input is $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)$, where each α_i is an unnormalized floating-point value with a $(p+2)$-bit signed exponent $\alpha_i.e \in [-2^{p-1} + 1, 2^p - 1)$ and an unnormalized (unsigned) mantissa $\alpha_i.m$ with bitwidth b, lower bound on MSNZB b', and scale sc such that $\alpha_i.m \in [2^{b'}, 2^b)$. That is, unlike a normalized mantissa, $\alpha_i.m$ can have at most $(b - b' - 1)$ leading 0's. The result of summation must be a normalized floating-point number with parameters (p,q), where $\mathsf{sc} \geqslant q$.

Our algorithm for generalized Summation follows the blueprint of FPSum with the following modifications to deal with unnormalized mantissas. First, after computing e_{max}, we set $e_{\mathsf{thr}} = e_{\mathsf{max}} - \mathsf{sc} - \tilde{n} - (b - b')$ so that we can still safely ignore the values smaller than the threshold $\Gamma = 2^{e_{\mathsf{thr}}}$. With this change, the maximum shift amount is $\mathsf{sc} + \tilde{n} + (b - b')$ (as opposed to $q + \tilde{n} + 1$) and consequently, we need to increase ℓ to $2b - b' + \mathsf{sc} + 2\tilde{n} + 1$ to ensure exact summation of n aligned and signed mantissas. Since normalization expects an input with scale q and the sum of aligned mantissas M has scale sc, we also change the scale of M to q and accordingly add $q - \mathsf{sc}$ to the exponent.

We describe our algorithm for generalized Summation, g-FPSum, parameterized by $b, b', \mathsf{sc}, p, q, n$ in Figure 3 and prove below that it achieves the same error bounds as before. Moreover, g-FPSum can easily be transformed to a 2PC protocol $\Pi_{\mathsf{g\text{-}FPSum}}^{b,b',\mathsf{sc},p,q,n}$ over secret shared input using the same steps as in Section 5.1.

Theorem 2. *The relative error of Figure 3 is at most* $\varepsilon + \delta\kappa + O(\varepsilon\delta\kappa)$, *where* $\varepsilon = 2^{-q-1}$, $\delta = 2^{-\mathsf{sc}-1}$, *and* $\kappa = \frac{\sum_{i \in [n]} |\alpha_i|}{\sum_{i \in [n]} \alpha_i}$ *is the condition number of summation.*

Proof. Let α_k be the element with the largest exponent, and let γ and γ' represent the sum of all elements and sum of all elements with exponent $\geqslant e_{\mathsf{thr}}$, respectively. We only need to argue that $|\gamma - \gamma'| \leqslant |\alpha_k| \cdot \delta$, where $\delta = 2^{-\mathsf{sc}-1}$, and the rest of proof follows in the same way as the proof of Theorem 1. It is easy to see that $|\alpha_k| \geqslant 2^{b'-s+e_{\mathsf{max}}}$ and each

```
Protocol $\Pi^{p,q,n}_{\mathsf{FPSum}}$
Input: $i \in [n], \langle \alpha_i \rangle^{\mathsf{FP}(p,q)}$.
Output: $\langle \gamma \rangle^{\mathsf{FP}(p,q)}$ s.t. $\gamma = \sum_{i=1}^{n} \alpha_i$.
 1: $\tilde{n} = \log(n); \ell = 2q + 2\tilde{n} + 3$
 2: Call $\langle e_{\mathsf{max}} \rangle^{p+2} = \Pi^{p+2,n}_{\mathsf{max}}(\{\langle \alpha_i.e \rangle^{p+2}\}_{i\in[n]})$   ▷ Find max exponent $e_{\mathsf{max}}$
 3: Set $\langle e_{\mathsf{thr}} \rangle^{p+2} = \langle e_{\mathsf{max}} \rangle^{p+2} - q - \tilde{n} - 1$   ▷ Threshold exponent $e_{\mathsf{thr}} = e_{\mathsf{max}} - q - \tilde{n} - 1$
    // Steps 5-11 implement steps 5,6,7 of Figure 1
 4: for $i \in [n]$ do
 5:     Call $\langle c_i \rangle^B = \Pi^{p+2}_{\mathsf{LT}}(\langle \alpha_i.e \rangle^{p+2}, \langle e_{\mathsf{thr}} \rangle^{p+2})$   ▷ Compare $\alpha_i$'s exponent with $< e_{\mathsf{thr}}$
 6:     Call $\langle m_i \rangle^{q+1} = \Pi^{q+1}_{\mathsf{MUX}}(\langle c_i \rangle^B, 0, \langle \alpha_i.m \rangle^{q+1})$   ▷ Set mantissa of $\alpha_i$ to 0 if $\alpha_i.e < e_{\mathsf{thr}}$
 7:     Set $\langle r_i \rangle^k = (\langle \alpha_i.e \rangle^{p+2} - \langle e_{\mathsf{thr}} \rangle^{p+2}) \bmod 2^k$, where $k = \lceil \log(q + \tilde{n} + 2) \rceil$   ▷ Compute shift amount $r_i \leqslant q + \tilde{n} + 1$ in $k$ bits
 8:     Call $\langle R_i \rangle^{q+\tilde{n}+2} = \Pi^{k,q+\tilde{n}+2}_{\mathsf{LUT}}(\mathsf{pow2}, \langle r_i \rangle^k)$   ▷ $R_i = 2^{r_i} = \mathsf{pow2}(r_i)$
 9:     Call $\left\langle m_i^{(\mathsf{align})} \right\rangle^{2q+\tilde{n}+2} = \Pi^{q+1,q+\tilde{n}+2,2q+\tilde{n}+2}_{\mathsf{UMult}}(\langle m_i \rangle^{q+1}, \langle R_i \rangle^{q+\tilde{n}+2})$   ▷ Align the mantissa $m_i$ by left-shifting by $r_i$
10:     Call $\left\langle m_i^{(\mathsf{ext})} \right\rangle^{\ell} = \Pi^{2q+\tilde{n}+2,\ell}_{\mathsf{ZXt}}\left(\left\langle m_i^{(\mathsf{align})} \right\rangle^{2q+\tilde{n}+2}\right)$   ▷ Extend to make space for addition of $n$ elements and sign-bit
11:     Call $\left\langle m_i^{(s)} \right\rangle^{\ell} = \Pi^{\ell}_{\mathsf{MUX}}(\langle \alpha_i.s \rangle^B, -1 \cdot \left\langle m_i^{(\mathsf{ext})} \right\rangle^{\ell}, \left\langle m_i^{(\mathsf{ext})} \right\rangle^{\ell})$   ▷ Set sign of mantissa same as input $\alpha_i$
12: Set $\left\langle M^{(s)} \right\rangle^{\ell} = \sum_{i\in[n]} \left\langle m_i^{(s)} \right\rangle^{\ell}$   ▷ Add the aligned mantissas to get $M^{(s)}$
13: Call $(\langle s \rangle^B, \langle z \rangle^B) = \Pi^{\ell}_{\mathsf{LT\&EQ}}(\left\langle M^{(s)} \right\rangle^{\ell}, 0)$   ▷ Set $s = \mathbf{1}\{M^{(s)} < 0\}, z = \mathbf{1}\{M^{(s)} = 0\}$
14: Call $\langle M \rangle^{\ell} = \Pi^{\ell}_{\mathsf{MUX}}(\langle s \rangle^B, -1 \cdot \left\langle M^{(s)} \right\rangle^{\ell}, \left\langle M^{(s)} \right\rangle^{\ell})$   ▷ $M = |M^{(s)}|$
15: Call $(\langle M_1 \rangle^{\ell}, \langle e_1 \rangle^{p+2}) = \Pi^{p,q,\ell-1}_{\mathsf{Normalize}}(\langle M \rangle^{\ell}, \langle e_{\mathsf{thr}} \rangle^{p+2})$   ▷ $e_{\mathsf{thr}}$ is the exponent for unnormalized $M$
16: Call $(\langle M_2 \rangle^{q+1}, \langle e_2 \rangle^{p+2}) = \Pi^{p,q,\ell-1}_{\mathsf{Round\&Check}}(\langle M_1 \rangle^{\ell}, \langle e_1 \rangle^{p+2})$   ▷ Reduce precision of mantissa from $(\ell - 1)$ to $q$
17: Call and return $\langle \gamma \rangle^{\mathsf{FP}(p,q)} = \Pi^{p,q}_{\mathsf{Clip}}(\langle z \rangle^B, \langle s \rangle^B, \langle e_2 \rangle^{p+2}, \langle M_2 \rangle^{q+1})$   ▷ Clip smaller than smallest representable values to 0
```

Figure 2: *Protocol for Floating-Point Summation.*

omitted term $\leqslant 2^{b-s+e_{\mathsf{thr}}-1}$. Thus, the sum of omitted terms has a magnitude $< n \cdot 2^{b-s+e_{\mathsf{thr}}-1} = 2^{b-s+e_{\mathsf{max}}-\mathsf{sc}-(b-b')-1} = 2^{b'-s+e_{\mathsf{max}}-\mathsf{sc}-1} \leqslant |\alpha_k| \cdot 2^{-\mathsf{sc}-1}$ and $|\gamma - \gamma'| \leqslant |\alpha_k| \cdot \delta$. □

5.3 Dot product and matrix multiplication

Given floating-point vectors $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)$ and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_n)$, DotProduct is defined as $\gamma = \sum_{i\in[n]} \alpha_i \cdot \beta_i$. Thus, dot product can be realized *naively* by first computing the intermediate products $\boldsymbol{\gamma} = \{\alpha_i \cdot \beta_i\}_{i\in[n]}$ using floating-point multiplication protocol for scalars, $\Pi^{p,q}_{\mathsf{FPMul}}$, from [60], and then calling our protocol for Summation, $\Pi^{p,q,n}_{\mathsf{FPSum}}$ (Figure 2) on $\boldsymbol{\gamma}$. We reduce the cost of this approach further by removing the normalization step in the floating-point multiplication followed by using our protocol for generalized Summation (Section 5.2). In our dot product protocol, the exponents of α_i and β_i are added and the mantissas of α_i and β_i are multiplied. The latter creates unnormalized $2q+2$-bit mantissas with scale $2q$ and values in $\in [1,4)$. We round these intermediate products to $q+2$ bits with scale q, perform clipping to 0 (to satisfy the constraints on exponents of inputs in generalized Summation in Section 5.2), and then invoke generalized summation ($\Pi^{q+2,q,q,p,q,n}_{\mathsf{g\text{-}FPSum}}$) to get the output of the dot product. Our protocol $\Pi^{p,q,n}_{\mathsf{FPDotProd}}$ for dot product is formally described in Figure 7, Appendix E. For $n = 1000$, our approach has $1.2\times$ lower communication than the naive approach and is just as precise as proved below. The protocols for matrix multiplication and convolutions build on top of dot product (Appendix F).

Theorem 3. *Our dot product protocol $\Pi^{p,q,n}_{\mathsf{FPDotProd}}$ is as precise as $\Pi^{p,q,n}_{\mathsf{FPSum}}(\{\Pi^{p,q}_{\mathsf{FPMul}}(\langle \alpha_i \rangle^{\mathsf{FP}(p,q)}, \langle \beta_i \rangle^{\mathsf{FP}(p,q)})\}_{i\in[n]})$.*

Proof. We first look at the worst-case relative error of $\Pi^{p,q,n}_{\mathsf{FPDotProd}}$. This protocol first computes the product of mantissas m_i exactly, and then rounds it to $q+2$ bits. Importantly, $q+2$ bits suffice for the output of rounding m'_i as $m_i \in [2^{2q}, (2^{q+1}-1)^2]$, and thus, $m'_i = \mathsf{RN}(m_i, q) \in [2^q, 2^{q+2}-2]$. Note that the rounding operation introduces a relative error of $|\delta_1| \leqslant \frac{2^{q-1}}{|m_i|} \leqslant 2^{-q-1}$ to the intermediate product γ_i. Unless γ_i is smaller than the smallest representable value, i.e., $\alpha_i.e + \beta_i.e < 1 - 2^{p-1}$, it is then input as is to $\Pi^{q+2,q,q,p,q,n}_{\mathsf{g\text{-}FPSum}}$, which adds a relative error of $|\delta_2| \leqslant \varepsilon(\kappa+1) + O(\varepsilon^2\kappa)$ to its output where $\varepsilon = 2^{-q-1}$ (Theorem 2). Thus, the absolute error of $\Pi^{p,q,n}_{\mathsf{FPDotProd}}$ is $|\sum_i(\gamma_i \cdot (1+\delta_2) - \alpha_i\beta_i)| =$

```
Algorithm g-FPSum^{b,b',sc,p,q,n}({α_i}_{i∈[n]}), sc ⩾ q
1:  $\tilde{n} = \log(n)$; $\ell = 2b - b' + \mathsf{sc} + 2\tilde{n} + 1$
2:  $e_{\mathsf{max}} = \mathsf{max}_i\ \alpha_i.e$
3:  $e_{\mathsf{thr}} = e_{\mathsf{max}} - \mathsf{sc} - \tilde{n} - (b - b')$
4:  for $i \in [n]$ do
5:      $m_i = (\alpha_i.e < e_{\mathsf{thr}})\ ?\ 0 : \alpha_i.m$
6:      $m_i^{(\mathsf{align})} = m_i \ll (\alpha_i.e - e_{\mathsf{thr}})$
7:      $m_i^{(s)} = \alpha_i.s\ ?\ -m_i^{(\mathsf{align})} : m_i^{(\mathsf{align})}$
8:  $M^{(s)} = \sum_{i\in[n]} m_i^{(s)}$
9:  $(s,z) = \mathsf{LT\&EQ}^{\ell}(M^{(s)}, 0)$
10: $M = s\ ?\ -M^{(s)} : M^{(s)}$
11: $(e, M') = \mathsf{Normalize}^{p,q,\ell-1}(e_{\mathsf{thr}} + q - \mathsf{sc}, M)$
12: $(e_1, M_1) = \mathsf{Round\&Check}^{p,q,\ell-1}(e, M')$
13: Return $\mathsf{Clip}^{p,q}(z, s, e_1, M_1)$
```

Figure 3: *Generalized Floating-Point Summation.*

$|\sum_i \alpha_i\beta_i \cdot (1+\delta_1)(1+\delta_2) - \alpha_i\beta_i| \leqslant |\sum_i \alpha_i\beta_i \cdot (\varepsilon(\kappa+2) + O(\varepsilon^2(\kappa+1)))|$, which implies a worst-care relative error of $\varepsilon(\kappa+2)+O(\varepsilon^2(\kappa+1))$.

Now, we look at the worst-case error of the naïve solution. $\Pi^{p,q}_{\mathsf{FPMul}}$ introduces a worst-case relative error of ε to the intermediate products, the same as $\Pi^{p,q,n}_{\mathsf{FPDotProd}}$. It also clips intermediate products when $\alpha_i.e + \beta_i.e < 1 - 2^{p-1}$. The intermediate products in the naïve solution are then input to $\Pi^{p,q,n}_{\mathsf{FPSum}}$ which has the worst-case relative error of $\varepsilon(\kappa+1) + O(\varepsilon^2\kappa)$, the same as $\Pi^{q+2,q,q,p,q,n}_{\mathsf{g\text{-}FPSum}}$. Thus, $\Pi^{p,q,n}_{\mathsf{FPDotProd}}$ is as precise as the naïve solution. □

6 BFloat16 training

BFloat16 or BF16 is essentially a lower-precision version of FP32 with the same dynamic range: mantissa bits q are reduced from 23 to 7 and exponent bits $p = 8$ are the same. In this section, we discuss our techniques for secure implementation of BF16 that give performance improvements over BEACON's FP32 while being more precise than standard platforms for BF16. Recall that in all platforms, BF16 is used just as a data-storage format, i.e., the inputs and outputs to each layer of the model are stored as BF16, and the arithmetic is performed in higher precision FP32 (Section 1.1). Although we focus on BF16, our techniques generalize in a straightforward manner to other representations, e.g., TensorFloat which uses $q = 10$. Below, we discuss linear layers and defer non-linear layers to Appendix I.

6.1 Linear Layers

As discussed in Section 5, our protocols for linear layers build upon the protocol for dot product. Hence, we discuss our techniques for secure dot product over BF16. For vectors of size n, the naïve approach for dot product that converts to FP32 before computing works as follows:

1. Left-shift the mantissas of the BF16 input vectors by 16 bits to convert them into FP32 representation.
2. Invoke the dot-product protocol $\Pi^{8,23,n}_{\mathsf{FPDotProd}}$ (Section 5.3) on FP32 vectors.
3. Round the output mantissa obtained above by 16 bits using $\Pi^{8,7,23}_{\mathsf{Round\&Check}}$ to get the final BF16 output.

As can be seen easily, this protocol is at least as expensive as $\Pi^{8,23,n}_{\mathsf{FPDotProd}}$, i.e., dot product of length n over FP32. Another approach to compute a dot product over BF16, which is much more efficient, is to directly invoke $\Pi^{8,7,n}_{\mathsf{FPDotProd}}$ and return its output as the final output. However, this has much worse error compared to the first approach that works over higher precision (Section 1.1.2). We now describe our protocol that achieves the best of both worlds: it is only $< 30\%$ more expensive than $\Pi^{8,7,n}_{\mathsf{FPDotProd}}$, and has the same precision as the standard BF16.

If we look closely at the naïve solution, it is first left-shifting input mantissas by 16 bits each, then multiplying them, and finally, rounding the multiplication result by 23 bits to get the mantissas for the intermediate products, the least significant 9 bits of which are always 0. It is easy to see that this is quite wasteful, and we can instead simply multiply the input mantissas to get 16-bit mantissas with scale 14 for the intermediate products without losing precision. However, there is an issue with this change. Since the mantissas being added only have scale 14 instead of scale 23 in the naïve approach, the following generalized Summation would ignore more values than the naïve approach (e_{thr} depends on the scale sc and lower scale leads to higher e_{thr} and larger magnitude values being dropped from the sum). Hence, we fix this in the second step, by increasing the scale of mantissa (but not the bitwidth) by 9-bits and accordingly adding 9 to the exponent to account for the scale change, thereby obtaining an exact intermediate product in 16 bits and scale 23. These mantissas with higher scale are now fed into the generalized Summation protocol by invoking $\Pi^{16,14,23,8,7,n}_{\mathsf{g\text{-}FPSum}}$. Our approach is much more efficient as it avoids the expensive steps of multiplication on 24-bit inputs (Step 3) and rounding by 23-bits (Step 6) in $\Pi^{8,23,n}_{\mathsf{FPDotProd}}$, as well as operates on mantissas of 16-bits as opposed to 25 bits in generalized summation. Our BF16 dot-product protocol $\Pi^{n}_{\mathsf{FPDotProdBF16}}$ is described in Figure 4 and its precision is proved below. For secure matrix multiplications over BF16 we use the COT-optimization discussed in Appendix F in conjunction with our protocol for dot product over BF16.

Theorem 4. *The relative error of $\Pi^{n}_{\mathsf{FPDotProdBF16}}$ is at most $\delta\kappa + \varepsilon + O(\varepsilon\delta\kappa)$, where $\varepsilon = 2^{-8}$, $\delta = 2^{-24}$, and κ is the condition number.*

Proof. We first calculate the relative error of our protocol $\Pi^{n}_{\mathsf{FPDotProdBF16}}$. We note that the intermediate products are

```
Protocol $\Pi^{n}_{\text{FPDotProdBF16}}(\{\alpha_i,\beta_i\}_{i\in[n]})$
Input: $i\in[n], \langle\alpha_i\rangle^{\text{FP}(8,7)}, \langle\beta_i\rangle^{\text{FP}(8,7)}$.
Output: $\langle\gamma\rangle^{\text{FP}(8,7)}$ s.t. $\gamma=\sum_{i\in[n]}\alpha_i\cdot\beta_i$.
1: for $i\in[n]$ do
2:     Set $\langle e_i\rangle^{10}=\langle\alpha_i.e\rangle^{10}+\langle\beta_i.e\rangle^{10}+9$
3:     Call $\langle m_i\rangle^{16}=\Pi^{8,8,16}_{\text{UMult}}(\langle\alpha_i.m\rangle^{8},\langle\beta_i.m\rangle^{8})$
4:     Set $\langle s_i\rangle^{B}=\langle\alpha_i.s\rangle^{B}\oplus\langle\beta_i.s\rangle^{B}$
5:     Call $\langle z_i\rangle^{B}=\Pi_{\text{OR}}(\langle\alpha_i.z\rangle^{B},\langle\beta_i.z\rangle^{B})$
6:     Call $\langle c\rangle^{B}=\Pi^{10}_{\text{LT}}(\langle e_i\rangle^{10},-127)$
7:     Call $\langle m'_i\rangle^{16}=\Pi^{16}_{\text{MUX}}(\langle c\rangle^{B},0,\langle m_i\rangle^{16})$
8:     Call $\langle e'_i\rangle^{10}=\Pi^{10}_{\text{MUX}}(\langle c\rangle^{B},-127,\langle e_i\rangle^{10})$
9:     Set $\langle\delta_i\rangle^{\text{FP}'}=(\langle z_i\rangle^{B},\langle s_i\rangle^{B},\langle e'_i\rangle^{10},\langle m'_i\rangle^{15})$
10: Return $\langle\gamma\rangle^{\text{FP}(8,7)}=\Pi^{16,14,23,8,7,n}_{\text{g-FPSum}}(\{\langle\delta_i\rangle^{\text{FP}'}\})$
```

Figure 4: *Protocol for BF16 Dot Product.*

calculated exactly, unless they are much smaller than the smallest representable exponent, i.e., $\alpha_i.e+\beta_i.e+9<-127$. Hence, the relative error of our scheme is same as relative error introduced by generalized summation $\Pi^{16,14,23,8,7,n}_{\text{g-FPSum}}$, i.e., $\delta\kappa+\varepsilon+O(\varepsilon\delta\kappa)$ for $\varepsilon=2^{-8},\delta=2^{-24}$ by Theorem 2. □

Corollary 1. $\Pi^{n}_{\text{FPDotProdBF16}}$ *is more precise than the naïve solution that left-shifts input mantissas, performs* $\Pi^{8,23,n}_{\text{FPDotProd}}$ *and rounds by* $\Pi^{8,7,23}_{\text{Round\&Check}}$.

Proof. For comparing our error with the naïve solution, we first observe that it also computes products exactly modulo clipping small values to 0. However, the naïve solution clips values for which $\alpha_i.e+\beta_i.e<-127$. That is, our approach clips less number of elements before summation. Next, the relative error of the the naïve solution depends on relative errors of $\Pi^{25,23,23,8,23,n}_{\text{g-FPSum}}$ and $\Pi^{8,7,23}_{\text{Round\&Check}}$. Again by Theorem 2, relative error of $\Pi^{25,23,23,8,23,n}_{\text{g-FPSum}}$ is $\delta(\kappa+1)+O(\delta^2\kappa)$. Next, since $\Pi^{8,7,23}_{\text{Round\&Check}}$ introduces worst-case relative error of ε, the final relative error of the naïve solution is $\delta(\kappa+1)+\varepsilon+O((\varepsilon+\delta)\delta\kappa)$ using similar argument on combining relative errors as in proof of Theorem 3. This clearly shows that the relative error of the naïve solution is more than our protocol $\Pi^{n}_{\text{FPDotProdBF16}}$. □

7 2PC for training and security proofs

We note that all our protocols for linear layers (Section 5,6) and non-linear layers (Appendix H, I)satisfy the condition that parties/servers P_0 and P_1 start with secret shares of inputs and end up with secret shares of outputs. Using this, our 2-party protocol for end-to-end secure training works by putting together protocols for linear layers and non-linear layers as specified by the training algorithm for both the forward and the backward passes. As is standard, the security of our training algorithms can be argued in the *hybrid model* [13] by composing the building blocks and we defer the complete security proof to Appendix J.

8 Implementation

We have implemented BEACON as a library in C++ on top of SECFLOAT with 5k LOC. This library's API provides operators arising in secure training, e.g., matrix multiplications, convolutions with various paddings and strides, ReLU and Maxpool, softmax, loss functions like mean squared error (MSE) and cross-entropy, etc. For other operators, e.g., trigonometric sine, we use SECFLOAT [60] as is.

9 Evaluation

We provide empirical evidence for the claims in Section 1.1, i.e., BEACON outperforms state-of-the-art in secure floating-point by over 6× (Section 9.1), and achieves secure floating-point training with < 6× the latency of secure fixed-point training with KS22 [41] (Section 9.2).

Evaluation Setup. We perform our experiments in the LAN setup between two 2.35 GHz AMD processor 16-core machines with 64 GiB memory that are connected through a network with 10 Gbps bandwidth and 73 μs latency (measured through `netperf`). We measure both end-to-end runtime and communication, without assuming an offline phase (similar to prior works [5,33,43,45,60–62,64]). All experiments use 16 threads for BEACON as well as all the baselines.

Benchmarks. We use the MNIST-10 dataset that has 60,000 28×28 monochrome images and the CIFAR-10 dataset that has 50,000 32×32 colored RGB images. We consider the following training benchmarks: MNIST-Logistic [55] (a single-layer logistic classifier for MNIST), MNIST-FFNN [41,43,55] (a 3-layer feed forward neural network for MNIST), CIFAR-LeNet [43] (a CNN with 2 convolutions), and CIFAR-HiNet [31] (a CNN with three convolutions). The description of these models is present in [1,2].

We also use the following microbenchmarks (designed to have similar runtimes) which are commonly occurring computations in ML: Summation-2k (2000 summations over vectors of length 2000 each), DotProduct-1k (1000 inner products between vectors of length 1000), and MatMul-100 (multiplying a 100×100 matrix with another 100×100 matrix).

9.1 Secure training with BEACON

Figure 5 compares the time and communication of BEACON with SECFLOAT [60], the state-of-the-art in secure floating-point, on the microbenchmarks for linear layers (Figure 5(a)-5(c)) and the training benchmarks for a batch iteration (Figure 5(d)-5(g)). We relegate the evaluation of non-linear mi-

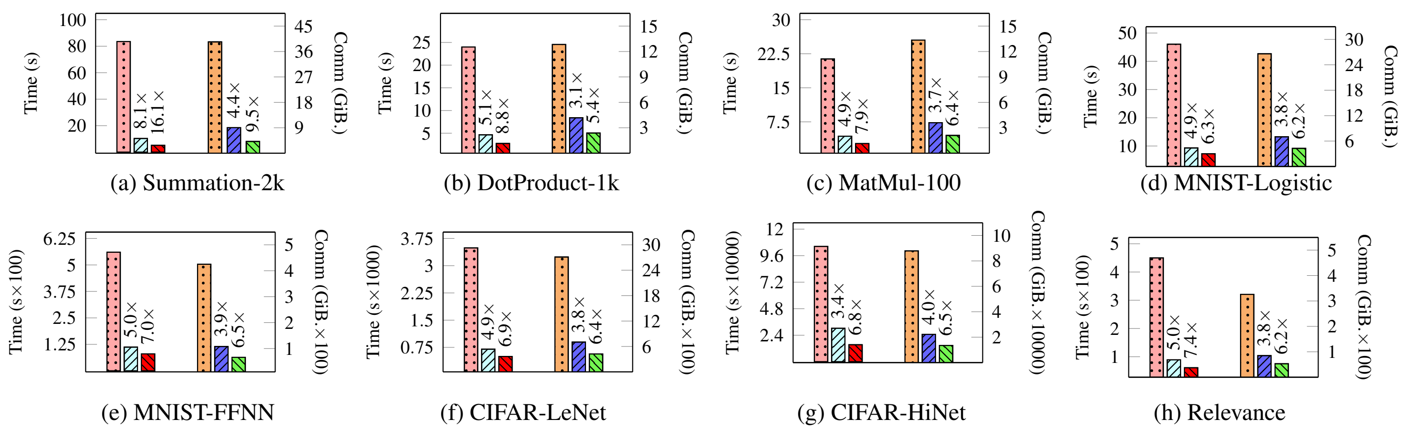


Figure 5: Performance comparison of SECFLOAT (time - ░, comm - 🟧) with BEACON FP32 (time - ▨, comm - 🟦) and BEACON BF16 (time - 🟥, comm - 🟩). The left bar group compares latency and the right group compares communication. Improvement factors of BEACON (both FP32 and BF16) are also shown.

crobenchmarks to Appendix C as 99% of secure training cost in SECFLOAT comes from linear layers (Appendix A). In addition to our training benchmarks, we also consider the Relevance model (Figure 7(h)), a benchmark proposed by SECFLOAT [60]. We observe that BEACON has $3.4-8.1\times$ lower latency and $3.1-4.4\times$ lower communication than SECFLOAT over FP32 tasks. The SECFLOAT baseline decomposes a compound operation into individual scalar operations that suffer from performance overheads caused by many normalization and rounding steps. When compared with SECFLOAT over BF16 tasks, BEACON improves latency by $6.3-16.1\times$ and the communication by $5.4-9.5\times$. Our further improvements with BF16 are again due to SECFLOAT's use of scalar operations which require that all arithmetic be performed in FP32 (Section 1.1.2).

Similar to KS22, we set the mini-batch size to 128 for all benchmarks except for Relevance, where SECFLOAT sets the mini-batch size to 32. Our evaluation in Figure 5 is for one training iteration with these mini-batch sizes. We observe that BEACON has $6.3-7.4\times$ lower latency and $6.2-6.5\times$ lower communication than the baseline on training tasks over BF16, thus demonstrating the performance benefits of our protocols.

9.2 Cost of BEACON vs. KS22

A curious reader might wonder about the overheads of running secure floating-point w.r.t. secure training with fixed-point approximations. In this section, we compare the training cost of one epoch of BEACON (over BF16) with the state-of-the-art secure fixed-point training framework KS22 [41]. We instantiated KS22 with the configuration from the paper, i.e., using 64-bit fixed-point and the default semi-honest secure 2PC backend (semi-homomorphic encryption or `hemi-party.x`).

Benchmark	Time (minutes)		Comm. (GiB)	
	BEACON	KS22	BEACON	KS22
MNIST Logistic	56	16 (3.3×)	2,011	58 (34.8×)
MNIST FFNN	626	106 (5.9×)	30,820	316 (97.5×)
CIFAR LeNet*	3,285	1,065 (3.1×)	165,594	7,881 (21×)
CIFAR HiNet*	100,827	32,137 (3.1×)	5,280,375	214,061 (24.7×)

Table 3: Time (in minutes) and communication (in GiB) per epoch of BEACON vs KS22 [41]. Numbers in parentheses represent the overhead of BEACON over KS22.
*: extrapolated from 10 iterations

Sometimes KS22 goes out of memory[9] when running a full epoch and in these cases we have extrapolated the time per epoch based on iterations that can run on our current set up. Table 3 summarizes the results. We found that the latency of BEACON is within $3-6\times$ of KS22 for our training benchmarks. The communication of BEACON, on the other hand, is $21-100\times$ higher than that of KS22. This is due to BEACON's use of oblivious transfers (OT) that are known to be communication heavy compared to homomorphic encryption based protocols used in KS22. Techniques such as Silent OT [11] can significantly reduce communication overheads of BEACON.

10 Related Work

Training algorithms. We have focused on FP32 and BF16 training. Other ML training algorithms include hybrid al-

[9]KS22 generates and stores pre-processing material that can overflow memory for large number of iterations.

gorithms that mix floating-point and integers. In quantized training, performance critical operations like matrix multiplications are performed on 8-bit/16-bit integers and precision critical operations like softmax or weight updates are performed in floating-point [8, 18, 20, 21, 34, 59, 71, 75, 76]. In block floating-point training, different layers use different scales and these scales are updated dynamically depending on the magnitudes of the runtime values [24, 68]. The advantage of this approach is that all weights of a layer share a common exponent and hence all matrix multiplications can be done over integers. Recently, 8-bit floating-point training is gaining traction [7, 52].

Defense against attacks like data poisoning [19] and techniques like differential privacy [25] are orthogonal to BEACON. These mitigations involve changing the training algorithms and BEACON is expressive enough to run the modified algorithms as well.

The techniques for training in the security literature can be classified as centralized, federated, and MPC-based.

Centralized. A centralized approach to secure training is for all the parties to give all their sensitive training data to a trusted hardware that does the floating-point computation in a trusted execution environment (TEE) and returns the result. However, the TEEs are susceptible to side-channel attacks [12, 30] and the use of secure 2PC makes BEACON provably resistant to them.

Federated. A well-known decentralized approach to multiparty training is through *federated learning* [50]. Here, multiple parties iteratively train in floating-point, aggregate their gradients, and update their models with the aggregated gradient. However, these leaked aggregated gradients have been used in various attacks to reveal information about the sensitive datasets [32, 51, 77].

MPC-based works. Kelkar et al. [39] run fixed-point training of poisson regression models with 2PC. Helen [74] provides stronger malicious security but is limited to the fixed-point training of linear classifiers. There are many works that use 2PC for the related problem of secure inference [10, 33, 35, 49, 53, 55, 58, 61, 62]. Prior works on secure floating-point in the honest majority setting include [6, 14, 15, 38, 42]. Other secure training works use threat models different from 2PC, e.g., honest majority [54, 64, 66, 67] and dealer-based [43, 69].

11 Conclusion

BEACON beats prior secure floating-point arithmetic work on ML tasks by over 6× while providing formal precision guarantees. There are three primary directions, orthogonal to this work, in which future research can extend BEACON, building upon our novel algorithms for precise compound operations. a) ImageNet-scale training is currently out of reach. Secure training on datasets with millions of images will require GPU support that BEACON lacks. b) Like all prior works on secure 2-party training, we have focused on security against semi-honest adversaries and would like to extend BEACON to provide security against active adversaries. One way to achieve this is by running our algorithms with MP-SPDZ as all our building blocks are easily supported by edabits [26]. c) Finally, we will like to improve the performance of BEACON by introducing a trusted dealer that provides correlated randomness, e.g., oblivious transfers.

References

[1] Deep learning training with multi-party computation. https://github.com/csiro-mlai/deep-mpc.

[2] Deep learning training with multi-party computation. https://github.com/csiro-mlai/deep-mpc/tree/more-models.

[3] EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[4] IEEE standard for floating-point arithmetic. *IEEE STD 754-2019 (Revision of IEEE 754-2008)*, 2019.

[5] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J. Kusner, and Adrià Gascón. QUOTIENT: two-party secure neural network training and prediction. In *CCS*, 2019.

[6] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS*, 2013.

[7] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2022.

[8] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *NIPS*, 2018.

[9] G. R. Blakley. Safeguarding cryptographic keys. In *International Workshop on Managing Requirements Knowledge*, 1979.

[10] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. MP2ML: a mixed-protocol machine learning framework for private inference. In *ARES*, 2020.

[11] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *CCS*, 2019.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX WOOT*, 2017.

[13] Ran Canetti. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptology*, 2000.

[14] Octavian Catrina. Evaluation of floating-point arithmetic protocols based on shamir secret sharing. In *ICETE (Selected Papers)*, 2019.

[15] Octavian Catrina. Performance analysis of secure floating-point sums and dot products. In *COMM*, 2020.

[16] Harsh Chaudhari, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively secure 4pc for secure training and inference. In *NDSS*, 2022.

[17] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS*, 2020.

[18] Xi Chen, Xiaolin Hu, Hucheng Zhou, and Ningyi Xu. Fxpnet: Training a deep convolutional neural network in fixed-point representation. *IJCNN*, 2017.

[19] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *CoRR*, abs/1712.05526, 2017.

[20] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *ArXiv*, abs/1602.02830, 2016.

[21] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesus Corbal, Nikita Shustrov, Roma Dubtsov, Evarist Fomenko, and Vadim Pirogov. Mixed Precision Training of Convolutional Neural Networks using Integer Operations, 2018.

[22] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In *CCS*, 2015.

[23] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the Communication Barrier in Secure Computation using Lookup Tables. In *NDSS*, 2017.

[24] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. In *NIPS*, 2018.

[25] Cynthia Dwork. Differential privacy: A survey of results. In *TAMC*, 2009.

[26] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO*, 2020.

[27] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Secure linear regression on vertically partitioned datasets. *IACR Cryptol. ePrint Arch.*, 2016.

[28] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 1991.

[29] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, 1987.

[30] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel SGX. In *EUROSEC*, 2017.

[31] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[32] Briland Hitaj, Giuseppe Ateniese, and Fernando Pérez-Cruz. Deep models under the GAN: information leakage from collaborative deep learning. In *CCS*, 2017.

[33] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure two-party deep neural network inference. In *USENIX Security Symposium*, 2022.

[34] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, 2018.

[35] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security Symposium*, 2018.

[36] William M. Kahan. Further remarks on reducing truncation errors. *Communications of the ACM*, 1965.

[37] Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training. *CoRR*, abs/1905.12322, 2019.

[38] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *Int. J. Inf. Sec.*, 2015.

[39] Mahimna Kelkar, Phi Hung Le, Mariana Raykova, and Karn Seth. Secure poisson regression. In *USENIX Security Symposium*, 2022.

[40] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, 2020.

[41] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In *ICML*, 2022.

[42] Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *FC*, 2016.

[43] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubhabrata Sengupta, Mark Ibrahim, and Laurens van der Maaten. CrypTen: Secure multi-party computation meets machine learning. In *NIPS*, 2021.

[44] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[45] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *IEEE S&P*, 2020.

[46] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[47] Jay P. Lim and Santosh Nagarakatte. RLIBM-32: high performance correctly rounded math libraries for 32-bit floating point representations. In *PLDI*, 2021.

[48] Yehuda Lindell. How to simulate it – a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography*, 2017.

[49] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *CCS*, 2017.

[50] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.

[51] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *IEEE S&P*, 2019.

[52] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart F. Oberman, Mohammad Shoeybi, Michael Y. Siu, and Hao Wu. FP8 formats for deep learning. *CoRR*, abs/2209.05433, 2022.

[53] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security Symposium*, 2020.

[54] Payman Mohassel and Peter Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*, 2018.

[55] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2017.

[56] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *CCS*, 2013.

[57] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE S&P*, 2013.

[58] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol secure Two-Party computation. In *USENIX Security Symposium*, 2021.

[59] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[60] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. SecFloat: Accurate Floating-Point meets Secure 2-Party Computation. In *IEEE S&P*, 2022. https://ia.cr/2022/322.

[61] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. SIRNN: A math library for secure inference of RNNs. In *IEEE S&P*, 2021.

[62] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow2: Practical 2-Party Secure Inference. In *CCS*, 2020.

[63] Adi Shamir. How to share a secret. *CACM*, 1979.

[64] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. Cryptgpu: Fast privacy-preserving machine learning on the GPU. In *IEEE S&P*, 2021.

[65] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*. Siam, 1997.

[66] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019.

[67] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETs*, 2021.

[68] Maolin Wang, Seyedramin Rasoulinezhad, Philip H. W. Leong, and Hayden Kwok-Hay So. NITI: training integer neural networks using integer-only arithmetic. *IEEE TPDS*, 2022.

[69] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU platform for secure computation. In *USENIX Security Symposium*, 2022.

[70] N. Whitehead and A. Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *nVidia technical white paper*, 2011.

[71] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. In *ICLR*, 2018.

[72] Andrew Yao. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS*, 1986.

[73] Andrew C. Yao. Protocols for secure computations. In *FOCS*, 1982.

[74] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

[75] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *ArXiv*, abs/1606.06160, 2016.

[76] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *ICLR*, 2017.

[77] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. In *NIPS*, 2019.

A Cost split between linear and non-linear layers with SECFLOAT

In Table 4, we show how the total runtime/communications of the training benchmark is divided between linear and non-linear layers for the prior state-of-the-art SECFLOAT. It is clearly seen that linear layers dominate in all DNNs consuming $> 99\%$ of both runtime/communications. Even for single layered MNIST-Logistic, linear layers contribute 96% to the runtime.

B Profiling of SECFLOAT's operations

In (Section 1.1), we claimed that over 82% of the runtime in SECFLOAT's addition operation was spent in normalization and rounding. To obtain this split (Table 5), we measured the runtime/communication for 10,000 instances of addition. A large number of instances $(10,000)$ was chosen because the design of SECFLOAT is inherently SIMD. Similarly, it was also claimed in (Section 2) that 85% of the runtime in matrix multiplication was spent in summations. This split (Table 6) is obtained by multiplying a 100×100-sized matrix with another 100×100-sized matrix, one of the microbenchmarks considered in Section 9.

C Evaluation on non-linear layers

Figure 6 shows empirical improvements of BEACON over SECFLOAT for Softmax-100 (1000 softmax over vectors of length 100 each) and Sigmoid-1m (pointwise sigmoid of a vector of 1 million elements). For ReLUs, BEACON's performance is similar to SECFLOAT. For sigmoid, BEACON's specialized BF16 protocol improves the performance by over $5\times$ compared to SECFLOAT. The improvement is much smaller for FP32 as BEACON uses SIMD FP32 exponentiations in this case. For Softmax-100, FP32 exponentiations are again the bottleneck for BEACON and SECFLOAT over both BF16 and FP32, and thus, the improvements from compound operations in BEACON lead to comparatively modest benefits. In fact, BF16 performance is even worse than FP32 in this case due to the FP32 to BF16 conversion required at the end.

D Multiple number representations

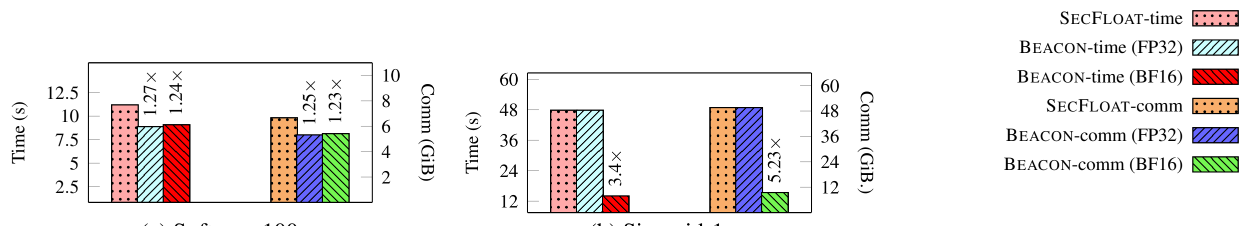
Table 7 shows how the performance of BEACON changes with the number representation used for data storage. We evaluate FP32 (IEEE's representation with 8-bit exponent and 23-bit mantissa supported in most CPUs), FP19 (NVIDIA's representation with 8-bit exponent and 10-bit mantissa supported in the latest GPUs), and BF16 (Google's representation with 8-bit exponent and 7-bit mantissa supported on the latest TPUs, some GPUs, and some CPUs). Our novel protocols ensure that the representations that require lower number of bits have better performance even though the underlying computation in each of these cases is required to be as precise as FP32. In particular, the performance of BF16 is about $2\times$ better than standard 32-bit floating-point.

E Dot Product Protocol $\Pi^{p,q,n}_{\mathsf{FPDotProd}}$

Our protocol for dot product is formally described in Figure 7.

	Time (s)			Comm (GiB)		
	Linear	Non-Linear	% Linear	Linear	Non-Linear	% Linear
MNIST-Logistic	44.2	1.8	95.96%	26.48	0.13	99.47%
MNIST-FFNN	558	1.97	99.65%	424.86	0.14	99.97%
Relevance	448	1.57	99.65%	325.03	0.07	99.98%
CIFAR-LeNet	3484	5.5	99.84%	2710.67	0.86	99.97%
CIFAR-HiNet	103572	899	99.14%	87787	122.8	99.86%

Table 4: Split in cost between linear and non-linear layers for SECFLOAT.



(a) Softmax-100 (b) Sigmoid-1m

Figure 6: Comparing the performance of SECFLOAT (dotted) with BEACON (striped, both FP32 and BF16) on non-linear functions. The left group of bars compares latency and the right compares communication (comm). The improvement factor of BEACON is also shown.

	Total	Norm.	% Norm.
Time (s)	0.384	0.316	82.3%
Comm (MiB)	105.58	81.28	76.9%

Table 5: Split of SECFLOAT's Addition. Norm. refers to normalization and rounding steps.

	Total	Summ.	% Summ.
Time (s)	20.24	17.35	85.7%
Comm (MiB)	13668	10784	78.9%

Table 6: Split of SECFLOAT's Matrix Multiplication. Summ. refers to Summation step.

Microbenchmark		Time (s)	Comm. (GiB)
Summation-2k	FP32	10.29	9.05
	FP19	5.83 (1.77×)	4.9 (1.85×)
	BF16	5.18 (1.99×)	4.15 (2.18×)
DotProduct-1k	FP32	4.65	4.17
	FP19	3.47 (1.34×)	2.94 (1.42×)
	BF16	2.74 (1.7×)	2.38 (1.75×)
MatMul-100	FP32	4.34	3.53
	FP19	2.94 (1.47×)	2.51 (1.41×)
	BF16	2.72 (1.59×)	2.1 (1.68×)

Table 7: Comparing cost of BEACON for various number representations. Improvement factor of FP19/BF16 over FP32 shown in parentheses.

F Matrix Multiplication

Matrix multiplication is a fundamental operation of ML training. Fully connected layers when operating on a mini-batch of images need to multiply a weight matrix with a matrix of images. Similarly, matrix multiplications are used to implement convolutions where a filter matrix is convolved with an input image. Finally, the implementations of transposed convolutions in PyTorch used for backpropagation over convolution layers also require matrix multiplication. We build these operators on top of dot product with additional optimizations.

The matrix multiplications arising from convolutions and transposed convolutions can have many entries that are statically known to be zeros (which arise because of padding). Matrix multiplications can be implemented using batched instances of many dot products that can use our protocol $\Pi_{\mathsf{FPDotProd}}$. Below, we discuss optimizations over this straightforward approach as well as how we can avoid multiplications with zeros by a careful decomposition.

Given floating-point matrices $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ of dimensions $n_1 \times n_2$ and $n_2 \times n_3$, matrix multiplication is defined as $\boldsymbol{\gamma} = \boldsymbol{\alpha} \times \boldsymbol{\beta}$, where $\gamma_{i,j} = \sum_{k \in [n_2]} \alpha_{i,k} \cdot \beta_{k,j}$ and $i \in [n_1], j \in [n_3]$. Thus, matrix multiplication is equivalent to $n_1 n_3$ dot products on vectors of size n_2. We further optimize the cost of matrix multiplication by taking advantage of the fact that each value in $\boldsymbol{\alpha}$ is being multiplied by n_3 values in $\boldsymbol{\beta}$. Thus, we can use the COT-batching trick [55] from the matrix-multiplication protocol of [61] to compute the product of mantissas of the input matrices. An important detail to note here is that we need to just compute the $n_1 n_2 n_3$ intermediate products and do not need to add them up. Thus, we do not extend one of the input integer matrices by $\log(n_2)$ bits before multiplication as in [61] to prevent overflows while adding these products. This optimization leads to great improvements in the computation

Protocol $\Pi^{p,q,n}_{\mathsf{FPDotProd}}$

Input: $i \in [n], \langle\alpha_i\rangle^{\mathsf{FP}(p,q)}, \langle\beta_i\rangle^{\mathsf{FP}(p,q)}$.
Output: $\langle\gamma\rangle^{\mathsf{FP}(p,q)}$ s.t. $\gamma = \sum_{i\in[n]} \alpha_i \cdot \beta_i$.

```
1:  for $i \in [n]$ do
2:      Set $\langle e_i\rangle^{p+2} = \langle\alpha_i.e\rangle^{p+2} + \langle\beta_i.e\rangle^{p+2}$
3:      Call $\langle m_i\rangle^{2q+2} = \Pi^{q+1,q+1,2q+2}_{\mathsf{UMult}}(\langle\alpha_i.m\rangle^{q+1}, \langle\beta_i.m\rangle^{q+1})$
4:      Set $\langle s_i\rangle^B = \langle\alpha_i.s\rangle^B \oplus \langle\beta_i.s\rangle^B$
5:      Call $\langle z_i\rangle^B = \Pi_{\mathsf{OR}}(\langle\alpha_i.z\rangle^B, \langle\beta_i.z\rangle^B)$
6:      Call $\langle m'_i\rangle^{q+2} = \Pi^{2q+2,q}_{\mathsf{RN}}(\langle m.i\rangle^{2q+2})$
7:      Call $\langle c\rangle^B = \Pi^{p+2}_{\mathsf{LT}}(\langle e_i\rangle^{p+2}, 1-2^{p-1})$
8:      Call $\langle m''_i\rangle^{q+2} = \Pi^{q+2}_{\mathsf{MUX}}(\langle c\rangle^B, 0, \langle m'_i\rangle^{q+1})$
9:      Call $\langle e'_i\rangle^{p+2} = \Pi^{p+2}_{\mathsf{MUX}}(\langle c\rangle^B, 1-2^{p-1}, \langle e_i\rangle^{p+2})$
10:     Set $\langle\delta_i\rangle^{\mathsf{FP}'} = (\langle z_i\rangle^B, \langle s_i\rangle^B, \langle e'_i\rangle^{p+2}, \langle m''_i\rangle^{q+2})$
11: Return $\langle\gamma\rangle^{\mathsf{FP}(p,q)} = \Pi^{q+2,q,q,p,q,n}_{\mathsf{g\text{-}FPSum}}(\{\langle\delta_i\rangle^{\mathsf{FP}'}\})$
```

Figure 7: *Protocol for Dot Product.*

of this step [55,61,62] and leads to an overall improvement in communication of 1.22× over naïve dot-product based solution for a matrix multiplication on matrices with dimensions 100×1000 and 1000×10.

Next, both convolutions as well transposed convolutions when converted to matrix multiplication can lead to varying number of entries that are statically 0 depending on the padding parameters. Intuitively, computing convolutions involves sliding filter over the padded image. To avoid multiplications with zeros, we divide the positioning of filter into 2 cases. First, where the entire filter is over the image (thus, no multiplications with zeros) and second, where the filter is put at the image boundary and involves a padding of zeros. We directly translate the first case as a matrix multiplication and implement it as described above. For the second case, we decompose the computation as a collection of dot products of varying sizes depending on the number of non-zero entries.

G Helper Protocols

G.1 Normalization

For a fixed-point value $x \in \mathbb{Z}_{2^\ell}$ with scale s, we use $[\![x]\!]_{\ell,s}$ to denote the corresponding real value. Normalization takes an unnormalized mantissa m of $Q+1$ bits with scale q and an exponent e in $p+2$ bits, and outputs a normalized mantissa m' in $Q+1$ bits with scale Q and an adjusted exponent e' such that $2^{e'} \cdot [\![m']\!]_{Q+1,Q} = 2^e \cdot [\![m]\!]_{Q+1,q}$. Let $m \in [2^k, 2^{k+1})$ for some $k \in [0,Q]$. The normalization works as follows: first compute $m' = m \ll (Q-k) \in [2^Q, 2^{Q+1})$ and set its scale to Q. As a result, we get a normalized mantissa with scale Q as expected from the output. To account for these changes to mantissa, the exponent needs to be adjusted. We add $Q-q$ to exponent and subtract $Q-k$ from it to account for the increase in scale and the left-shift, respectively. Combining the two operations, we get our output exponent $e' = e+k-q$. From these steps, it is easy to see that $2^{e'} \cdot [\![m']\!]_{Q+1,Q} = 2^e \cdot [\![m]\!]_{Q+1,q}$. The normalization protocol is described in Figure 8.

Protocol $\Pi^{p,q,Q}_{\mathsf{Normalize}}$

Input: Unnormalized $\left(\langle m\rangle^{Q+1}, \langle e\rangle^{p+2}\right)$, scale of m is q.
Output: Normalized $\left(\langle m'\rangle^{Q+1}, \langle e'\rangle^{p+2}\right)$.

```
1: Call $\langle k\rangle^{Q+1}, \langle K\rangle^{Q+1} = \Pi^{Q+1}_{\mathsf{MSNZB}}(\langle m\rangle^{Q+1})$
2: Call $\langle m'\rangle^{Q+1} = \Pi^{Q+1,Q+1,Q+1}_{\mathsf{UMult}}(\langle m\rangle^{Q+1}, \langle K\rangle^{Q+1})$
3: Set $\langle e'\rangle^{p+2} = \langle e\rangle^{p+2} + (\langle k\rangle^{Q+1} \bmod 2^{p+2}) - q$
4: Return $(\langle m'\rangle^{Q+1}, \langle e'\rangle^{p+2})$
```

Figure 8: *Normalize mantissa and adjust exponent accordingly*

Protocol $\Pi^{p,q,Q}_{\mathsf{Round\&Check}}$

Input: High-precision $\langle m\rangle^{Q+1}$ and $\langle e\rangle^{p+2}$.
Output: Low-precision $\langle m'\rangle^{q+1}$ and $\langle e'\rangle^{p+2}$.

```
1: Call $\langle c\rangle^B = \Pi^{Q+1}_{\mathsf{LT}}(\langle m\rangle^{Q+1}, 2^{Q+1} - 2^{Q-q-1})$.
2: Call $\langle m_c\rangle^{q+1} = \Pi^{Q+1,Q-q}_{\mathsf{RN}}(\langle m\rangle^{Q+1})$
3: Call $\langle m'\rangle^{q+1} = \Pi^{q+1}_{\mathsf{MUX}}(\langle c\rangle^B, \langle m_c\rangle^{q+1}, 2^q)$.
4: Call $\langle e'\rangle^{p+2} = \Pi^{p+2}_{\mathsf{MUX}}(\langle c\rangle^B, \langle e\rangle^{p+2}, \langle e\rangle^{p+2} + 1)$.
5: Return $(\langle m'\rangle^{q+1}, \langle e'\rangle^{p+2})$
```

Figure 9: *Round mantissa and check for its overflow*

G.2 Round and Check

The protocol for Round&Check is described in Figure 9. The normalization check is quite simple: since we know that the only $m \in [2^{Q+1} - 2^{Q-q-1}, 2^{Q+1})$ lead to the one and only unnormalized output $m' = 2^{q+1}$, we can simply check for this condition and set the output accordingly. In particular, instead of outputting $m' = 2^{q+1}, e' = e$ in this case, we output $m' = 2^q, e' = e+1$ which doesn't introduce any error and thus, the final output has relative error $\leqslant \varepsilon$.

G.3 Clip details

For target representation (p,q) and input α, the clipping protocol $\Pi^{p,q}_{\mathsf{Clip}}$ is easily realized by first performing a comparison $\langle c\rangle^B = \Pi^{p+2}_{\mathsf{LT}}(\langle\alpha.e\rangle^{p+2}, -2^{p-1}+2)$ of $p+2$ bits to check if exponent is less than or equal to the smallest representable exponent, and then setting mantissa to 0 and exponent to $-2^{p-1}+1$ in case $c = 1$ using two multiplexer operations: $\Pi^{q+1}_{\mathsf{MUX}}(\langle c\rangle^B, 0, \langle\alpha.m\rangle^{q+1})$ and $\Pi^{p+2}_{\mathsf{MUX}}(\langle c\rangle^B, -2^{p-1}+1, \langle\alpha.e\rangle^{p+2})$, respectively. Since the comparison operation is signed, we require $|\alpha.e| < 2^{p+1}$ for its correctness.

H Non-Linear Layers for FP32

Non-linear layers require both compound operations and SIMD operations. BEACON provides novel protocols for compound operations. For SIMD operations, we use the existing state-of-the-art protocols given by SECFLOAT [60] for pointwise multiplication, addition, division, comparison and exponentiation.

H.1 ReLU

$ReLU(x) = \max(x, 0)$ can simply be computed as a multiplexer over the sign bit.

H.2 Softmax

Given vector $\boldsymbol{\alpha} \in \mathbb{R}^n$ as input, softmax outputs a vector $\boldsymbol{\delta}$ such that $\delta_i = \frac{e^{\alpha_i}}{\sum_{j\in[n]} e^{\alpha_j}}$. In practice to avoid overflows in exponentiation, the maximum element is subtracted from every element of the array to create $\boldsymbol{\beta}$, i.e., $\beta_i = \alpha_i - \alpha'$, where $\alpha' = \mathsf{max}_j\, \alpha_j$. At a high level, softmax has the following steps:

1. Compute maximum element α' and subtract it from every vector element to get the vector $\boldsymbol{\beta}$ (with all negative entries).
2. Compute exponentiation on $\boldsymbol{\beta}$ to get $\boldsymbol{\gamma}$.
3. Sum the elements of $\boldsymbol{\gamma}$ to get θ.
4. Divide $\boldsymbol{\gamma}$ by θ and output the resulting vector.

All the above steps can be implemented using operations provided in SECFLOAT [60].

We improve upon this solution by optimizing steps 3 and 4. First, we use vector sum (Section 5.1) in step 3, which not only improves the efficiency, but also the accuracy of this step. Next, we observe that in step 4, all vector elements are being divided by the same value θ and we can get better amortization cost for this operation. In more detail, computation of output mantissa in the division functionality from [60] involves first approximating the reciprocal of divisor, followed by a multiplication with dividend and a rounding operation. Since the divisor is the same in all of the division operations, the reciprocal computation can be done just once.

H.3 Sigmoid/Tanh

For $\alpha \in \mathbb{R}$, sigmoid is defined as $\mathsf{Sigmoid}(\alpha) = \frac{1}{1+e^{-\alpha}}$. Clearly, this is equivalent to a softmax on vector of length 2, i.e., softmax on $\boldsymbol{\beta} = [0, \alpha]$.

We use $\mathsf{Tanh}(\alpha) = \frac{e^{\alpha}-e^{-\alpha}}{e^{\alpha}+e^{-\alpha}} = 2\cdot\mathsf{Sigmoid}(2\alpha) - 1$.

I Non-Linear Layers for BF16

The naïve approach to compute non-linear layers is also to first convert to FP32, then perform the operation, and finally round the output to BF16. In this section, we propose protocols that greatly improve upon this approach for some layers.

I.1 Sigmoid/Tanh

We observe that for layers with single-input operations like sigmoid and tanh, we can avoid all expensive computation while guaranteeing the correct output by simply using a lookup table. However, using a lookup table with 2^{16} entries of 16-bits each is not an efficient solution, as it requires communicating 128 KiB per operation which is even much more expensive than the naïve solution. Instead, we take advantage of the domain knowledge of the operation to greatly reduce the size of the lookup table needed. In particular, for functions like sigmoid and tanh, and in general for activation functions used in machine learning, the output only varies significantly for a small domain $\mathcal{D}$ of inputs, and in the rest of the domain, the difference in real outputs is small enough that it leads to the same output over BF16. Thus, we can handle inputs $\in \mathcal{D}$ using a lookup table, and the rest of the inputs can be easily handled as a special case. Applying this strategy to sigmoid, we found that $\mathcal{D} = \{\alpha : |\alpha| \in [2^{-8}, 93)\}$, which requires just 4 bits of exponent to represent. Thus, it allows us to use a lookup table of size 2^{12} to handle all inputs in $\mathcal{D}$, and the output of lookup table can be 15-bits long as the sign-bit of output is always 0. Correct BF16 output for $|\alpha| < 2^{-8}$ is 0.5, for $\alpha \geqslant 93$ is 1 and for $\alpha \leqslant -93$, is 0. We handle these cases separately with multiplexers and use the floating-point comparison protocol from [60] for checking if $|\alpha| < 93$. Comparison of $|\alpha|$ with 2^{-8} can be done using the exponent alone and does not require a full fledged floating-point comparison. Our BF16 sigmoid protocol is given in Figure 10. Note that the $\Pi^{8,7}_{\mathsf{FPLT}}$ protocol compares two BF16 numbers and returns a boolean which is true iff the first argument is smaller [60]. Similar to sigmoid, we can also get the tanh BF16 protocol, which also uses a lookup table of size 2^{12}. Compared to the naïve approach, our lookup table based solution is around $6\times$ more communication efficient.

I.2 Softmax

For softmax, we simply use our softmax over FP32 (Appendix H.2). The lookup table based approach can't be used to reduce costs as even the first operation in softmax, i.e. subtracting all vector elements by the maximum, is a two-input operation which would require a lookup table with 2^{32} entries which is very inefficient. We can't use lookup tables at any subsequent step as well because all intermediate values will be in 32-bits.

Protocol $\Pi_{\mathsf{FPSigmoidBF16}}$

Input: $\langle\alpha\rangle^{\mathsf{FP}(8,7)}$.

Output: $\langle\gamma\rangle^{\mathsf{FP}(8,7)}$ s.t. $\gamma = \mathsf{Sigmoid}(\alpha)$.

```
1: Set $\langle \mathsf{idx}_1\rangle^5 = \langle \alpha.s\rangle^B \,||\, ((\langle\alpha.e\rangle^{10} + 8) \bmod 2^4)$.
2: Set $\langle \mathsf{idx}_2\rangle^7 = \langle\alpha.m\rangle^8 \bmod 2^7$.
3: Set $\langle \mathsf{idx}\rangle^{12} = \langle \mathsf{idx}_1\rangle^5 \,||\, \langle \mathsf{idx}_2\rangle^7$.
4: Call $(\langle e\rangle^8, \langle m\rangle^7) = \Pi^{12,15}_{\mathsf{LUT}}(L_{\mathsf{Sigmoid}}, \langle \mathsf{idx}\rangle^{12})$.
5: Call $\langle e\rangle^{10} = \Pi^{8,10}_{\mathsf{ZXt}}(\langle e\rangle^8)$.
6: Call $\langle m\rangle^8 = \Pi^{7,8}_{\mathsf{ZXt}}(\langle m\rangle^7) + 2^7$.
7: Set $\langle\gamma'\rangle^{\mathsf{FP}(8,7)} = (0, 0, \langle e\rangle^{10}, \langle m\rangle^7)$.
8: Set $\langle\beta\rangle^{\mathsf{FP}(8,7)} = (\langle\alpha.z\rangle^B, 0, \langle\alpha.e\rangle^{10}, \langle\alpha.m\rangle^8)$
9: Call $\langle c_1\rangle^B = \Pi^{8,7}_{\mathsf{FPLT}}(\langle\beta\rangle^{\mathsf{FP}(8,7)}, 93)$.
10: Call $\langle c_2\rangle^B = \Pi^{10}_{\mathsf{LT}}(\langle\alpha.e\rangle^{10}, -8)$.
11: Call $\langle\eta\rangle^{\mathsf{FP}(8,7)} = \Pi^{20}_{\mathsf{MUX}}(\langle\alpha.s\rangle^B, 0.0, 1.0)$.
12: Call $\langle\gamma'\rangle^{\mathsf{FP}(8,7)} = \Pi^{20}_{\mathsf{MUX}}(\langle c_1\rangle^B, \langle\gamma'\rangle^{\mathsf{FP}(8,7)}, \langle\eta\rangle^{\mathsf{FP}(8,7)})$.
13: Call $\langle\gamma\rangle^{\mathsf{FP}(8,7)} = \Pi^{20}_{\mathsf{MUX}}(\langle c_2\rangle^B, 0.5, \langle\gamma'\rangle^{\mathsf{FP}(8,7)})$.
14: Return $\langle\gamma\rangle^{\mathsf{FP}(8,7)}$.
```

Figure 10: *Protocol for BF16 Sigmoid.*

J Security proofs

Since all our protocols are symmetric, let P_0/S_0 be the corrupted party without loss of generality. As is standard, we first argue the security of our protocols against a semi-honest adversary in the *hybrid model* [13] starting with compound operations, and then with similar arguments, we will prove the security of our overall protocol for secure training in the two-party (Appendix J.1) and client-server setting (Appendix J.2).

In particular, we first prove the security of our compound operation protocols in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model, where $\mathcal{F}_{\mathsf{BB}}$ is a collection of ideal functionalities for the building blocks (Table 1) and helper algorithms (Section 4.4). Note that the functionalities of all building blocks and helper algorithms maintain the invariant that both inputs and output are 2-out-of-2 secret shares. Now, we first look at our protocol for summation $\Pi^{p,q,n}_{\mathsf{FPSum}}$ (Figure 2). The ideal functionality $\mathcal{F}^{p,q,n}_{\mathsf{FPSum}}$ for summation is as follows: it takes 2-out-of-2 secret shares of the vector $\langle\alpha\rangle^{\mathsf{FP}(p,q)} = \{\langle\alpha_i\rangle^{\mathsf{FP}(p,q)}\}_i$ of floating-point values from P_0 and P_1, reconstructs α, computes the summation function on it to get γ, and finally, secret shares γ and sends the corresponding shares $\langle\gamma\rangle^{\mathsf{FP}(p,q)}$ to the two parties. To prove the security of $\Pi^{p,q,n}_{\mathsf{FPSum}}$ in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model w.r.t. $\mathcal{F}^{p,q,n}_{\mathsf{FPSum}}$, we first replace all calls to the building blocks in this protocol with their corresponding ideal functionalities to get ${}^*\Pi^{p,q,n}_{\mathsf{FPSum}}$. It is easy to see that ${}^*\Pi^{p,q,n}_{\mathsf{FPSum}}$ only has local computations and calls to ideal functionalities in $\mathcal{F}_{\mathsf{BB}}$. Thus, only outputs of these ideal functionalities make the adversary's view. Now, we describe the simulator $\mathcal{S}$ for ${}^*\Pi^{p,q,n}_{\mathsf{FPSum}}$: $\mathcal{S}$ sends the inputs of corrupted P_0 to $\mathcal{F}^{p,q,n}_{\mathsf{FPSum}}$ and sets the output of calls to functionalities in $\mathcal{F}_{\mathsf{BB}}$ as random values from appropriate domains, except the final call whose output is set such that P_0's output would be the same as the output received from $\mathcal{F}^{p,q,n}_{\mathsf{FPSum}}$, i.e., $\mathcal{S}$ sets the output of $\mathcal{F}^{p,q}_{\mathsf{Clip}}$ to $\langle\gamma\rangle_0^{\mathsf{FP}(p,q)}$. It is easy to see that the environment can't distinguish the joint distribution of P_0's view and the outputs of both parties in the ideal and real world. It follows directly from the security of secret sharing that the view of P_0 in this simulation is indistinguishable from its view in ${}^*\Pi^{p,q,n}_{\mathsf{FPSum}}$. The output distributions are independent of P_0's view and are also identical as they consists of two random sets of shares that have the same correlation owing to the correctness of ${}^*\Pi^{p,q,n}_{\mathsf{FPSum}}$, i.e., they represent γ on reconstruction. Thus, ${}^*\Pi^{p,q,n}_{\mathsf{FPSum}}$ is secure in the $\mathcal{F}_{\mathsf{BB}}$-hybrid model w.r.t. $\mathcal{F}^{p,q,n}_{\mathsf{FPSum}}$, and so is $\Pi^{p,q,n}_{\mathsf{FPSum}}$ by instantiating the functionalities in $\mathcal{F}_{\mathsf{BB}}$ with the corresponding secure protocols from Section 3.4 [13].

We can use the same argument to argue the security of all our protocols for compound operations for linear layers (Section 5,6) and non-linear layers (Appendix H, I). Let $\mathcal{F}_{\mathsf{Layers}}$ denote the ideal functionalities for all layers involved in the training protocol.

J.1 Two-party Setting.

Our two-party setting training protocol Π_{2PC} proceeds as follows: P_0 and P_1 secret-share their inputs, sequentially make calls to protocols for functionalities in $\mathcal{F}_{\mathsf{Layers}}$ according to the training algorithm M, and at the end of the protocol, exchange their output shares to learn the trained model. It is easy to argue the security of this protocol in the $\mathcal{F}_{\mathsf{Layers}}$-hybrid model w.r.t. the training ideal functionality $\mathcal{F}_{\mathsf{2PC}}$, which takes inputs x_0 and x_1 from P_0 and P_1, respectively, and outputs the trained model $y = M(x_0, x_1)$ to both parties. Note that after replacing the calls to protocols in Π_{2PC} with ideal functionalities in $\mathcal{F}_{\mathsf{Layers}}$, the only interaction between P_0 and P_1 is for input sharing at the beginning and at the end to reconstruct the training output. Thus, the simulator $\mathcal{S}$ can just send P_0's input to $\mathcal{F}_{\mathsf{2PC}}$, give random values as shares from P_1, provide random values from appropriate domains as the output of all functionalities in $\mathcal{F}_{\mathsf{Layers}}$, and adjust the final message sent to P_0 at the end to ensure P_0's output is y, which $\mathcal{S}$ learns as output from $\mathcal{F}_{\mathsf{2PC}}$. $\mathcal{S}$ can do so because it knows the final shares of P_0, which are only a function of P_0's input and the outputs it receives from functionalities in $\mathcal{F}_{\mathsf{Layers}}$ that are set by $\mathcal{S}$. Using similar arguments as before, it is easy to show that the joint distribution of P_0's view and P_1's output is indistinguishable in the ideal and real world, and thus, Π_{2PC} is secure w.r.t. $\mathcal{F}_{\mathsf{2PC}}$.

J.2 Client-Server model.

We have n clients $\{C_i\}_{i\in[n]}$ and two servers S_0 and S_1 in our training protocol Π_{CSM} in the client-server model. In this protocol, each client C_i first secret-shares its input x_i between the two servers. Then, the servers sequentially make calls to protocols for functionalities in $\mathcal{F}_{\mathsf{Layers}}$ according to the training algorithm M, and then send their output shares to the clients. Finally, the clients reconstruct the shares received from the servers to learn the trained model. In this section, we prove the security of this protocol w.r.t. the following ideal functionality $\mathcal{F}_{\mathsf{CSM}}$: $\forall i \in [n]$, takes input x_i from C_i, computes $y = M(x_1, \ldots, x_n)$ and sends it to all clients C_i.

We consider the general case where the adversary can corrupt any *strict subset* of clients (this includes no corrupted clients) and one of the servers, say S_0. There are two cases:

No client is corrupt: In this case, only S_0 is corrupt who has no input and no output in the ideal world. Hence, the simulation is straightforward by providing random values as shares of client's inputs and outputs of functionalities.

At least one client is corrupt: For this case, the simulation strategy is as follows: the simulator $\mathcal{S}$ sends the inputs from corrupted clients to $\mathcal{F}_{\mathsf{CSM}}$, receives S_1's share for all corrupted clients from the adversary, sends random shares to S_0 for honest clients inputs, provides random values from appropriate domains as the output of all functionalities in $\mathcal{F}_{\mathsf{Layers}}$, and finally, adjusts the final message sent to all corrupted clients at the end to ensure their output is y, which $\mathcal{S}$ learns as output from $\mathcal{F}_{\mathsf{CSM}}$. $\mathcal{S}$ can adjust the final message because it knows S_0's final shares which are only a function of client's input shares for S_0[10] and outputs of functionalities in $\mathcal{F}_{\mathsf{Layers}}$, all of which are known by $\mathcal{S}$. It is easy to show that the joint distribution of the view of the adversary and honest clients' output in this simulation is indistinguishable from Π_{CSM}. The corrupted server sees uniformly random elements in both worlds, and corrupted client's view only consists of random shares with the correlation that they reconstruct to y, which is the same in both cases owing to the correctness of our training protocol. Thus, our training protocol Π_{CSM} is secure w.r.t. $\mathcal{F}_{\mathsf{CSM}}$.

[10]Since $\mathcal{S}$ knows the inputs for corrupted clients as well as their input shares for S_1, it can determine the input shares for S_0 from corrupted clients.