Secure Floating-Point Training
Deevashwer Rathee, University of California, Berkeley; Anwesh Bhattacharya, Divya Gupta, and Rahul Sharma, Microsoft Research; Dawn Song, University of California, Berkeley
https://www.usenix.org/conference/usenixsecurity23/presentation/rathee
Secure Floating-Point Training

Deevashwer Rathee\textsuperscript{1}, Anwesh Bhattacharya\textsuperscript{2}, Divya Gupta\textsuperscript{2}, Rahul Sharma\textsuperscript{2}, and Dawn Song\textsuperscript{1}

\textsuperscript{1}University of California, Berkeley
\textsuperscript{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\textsuperscript{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.

\textsuperscript{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\textsuperscript{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\textsuperscript{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 [16,17,41,43,54,55,56,64,66,67]. 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,

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

\textsuperscript{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 BEACON⁴ 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× over the state-of-the-art libraries for 2PC of floating-point (Figure 5) and has < 6× 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× less precise⁵ 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 representations 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-

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

⁵A floating-point representation with \( q \)-bit mantissa 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 sum-ming 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 TensorFlow 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 $\sum_{i=0}^{n} y_i$), 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 $\epsilon \kappa n$ where $\epsilon$ is machine epsilon [28] of the floating-point representation and $\kappa$ 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 $\epsilon \kappa$. With this error in mind, we carefully determine a threshold such that we can pick a reasonable bitwidth $\ell$ 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 $\epsilon \kappa$ error. After doing summation of values that matter in $\ell$ bits, we perform one normalization and rounding. The final rounding leads to additional $\epsilon$ error and our overall relative error is at most $\epsilon \kappa + 1$. 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×.

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, …, 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 I{P} returns 1 if the predicate P is true and 0 otherwise; \( x|y \) concatenates bit-strings x and y. An ℓ-bit integer \( x \in \mathbb{Z}_\ell \) can be interpreted as an unsigned integer \( \text{uint}(x) = \zeta_f(x) \), where \( \zeta_f \) is a lossless lifting operator from bitstrings in \( \mathbb{Z}_\ell \) to integers in \( \mathbb{Z} \). Using the 2’s complement encoding, x can also be interpreted as a signed integer \( \text{int}(x) = \text{uint}(x) - \text{MSB}(x) \cdot 2^\ell \), where \( \text{MSB}(x) \) is the most-significant bit of x.

Fixed-point: An unsigned fixed-point number \( x \in \mathbb{Z}_\ell \) with bitwidth ℓ and scale s represents the real number \( \frac{\text{uint}(x)}{2^s} \in \mathbb{Q} \).

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 \( x \in \mathbb{Z}_\ell \) is split into two shares \( \langle x \rangle_0, \langle x \rangle_1 \in (\mathbb{Z}_\ell)^2 \) and party \( P_i \) holds \( \langle x \rangle_i \). Each secret share \( \langle x \rangle_i \) 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: \( x = \langle x \rangle_0 + \langle x \rangle_1 \mod 2^\ell \). 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 \( y = f(x_1, \ldots, x_n) \). 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 (ℓ − s)-bit value. Note that SECFFLOAT [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 \( \text{RN}(x,s) = \left\lfloor \frac{x}{2^s} \right\rfloor \), that does round to nearest and ties are always rounded up suffices and is also cheaper in 2PC. It is easy to see that \( \left\lfloor \frac{x}{2^s} \right\rfloor = \left\lfloor \frac{x + 2^s - 1}{2^s} \right\rfloor \), and hence, we implement our protocol as \( \Pi_{\text{RN}}(\langle x \rangle_i) = \Pi_{\text{TR}}(\langle x \rangle_i + 2^{s-1}) \), where \( \Pi_{\text{TR}} \) 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.
Floating-point operations, as defined by the IEEE-754 standard, are designed to ensure consistency and accuracy across different computing platforms. This standard provides a framework for handling floating-point numbers, including the representation of special values like NaNs and infinities, and the computation of arithmetical operations. The IEEE-754 standard specifies that operations are performed with a certain level of precision, known as the machine epsilon, which is the smallest positive number that can be added to 1 to yield a result different from 1. This epsilon is defined as $2^{-p}$, where $p$ is the number of bits in the floating-point format.

In the context of PyTorch, the precision of floating-point operations is crucial for ensuring the accuracy of computations, especially in scenarios where the correctness of the output is paramount. The implementation of floating-point operations in PyTorch must be designed to adhere to the IEEE-754 standard, ensuring that the difference between the ideal result and the actual output is as small as possible.

To 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 of the output. 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 $\alpha$ 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 $a$ and $r$ is $|a-r|/|r|$. 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 in PyTorch and BEACON, and performing several operations in sequence can well approximate 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 $\alpha$ parameterized by $p, q \in \mathbb{Z}^+$ as a 4-tuple $(\alpha.z, \alpha.s, \alpha.e, \alpha.m)$ where $\alpha.z = 1$ if $\alpha = 0$ is the zero-bit, $\alpha.s \in \{0, 1\}$ is the sign-bit (set if $\alpha \leq 0$), $\alpha.e \in \{0, 1\}^{p+2}$ is the unbiased exponent 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 $\alpha$ is parameterized by $\alpha.z = 1$ and the leading $-1$ bits if $\alpha.z = 0$. Identical to [60], we represent a floating-point value $\langle \alpha \rangle_p.q = ((\alpha.z)^B, (\alpha.s)^B, (\alpha.e)^{p+2}, (\alpha.m)^q)$. Finally, the Real value of $\alpha$, i.e., $\lceil \alpha \rceil_z$ is zero if $\alpha.z = 1$ and $\lceil -1 \alpha.s \cdot 2^{\text{int}((\alpha.e) \cdot \text{uint}(\alpha.m))} \rceil_{2^q}$ 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 F).

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 $\Pi^{\text{Normalize}}_{p,q}$. 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 $\varepsilon = 2^{-q-1}$. However, simply using $\Pi^{\text{Round}}_{Q^{Q+1}}$ 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 $\Pi^{\text{Round&Check}}_{p,q}$ that additionally performs this check and adjusts the exponent accordingly (Appendix F.2).

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 $\Pi^{\text{Clip}}_{p,q}$ sets inputs with exponent $< -126$ to 0 (Appendix F.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, where $p = 8, q = 23$. We defer the discussion of non-linear operations to Appendix G (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 $\alpha = (\alpha_1, \ldots, \alpha_n)$, where each $\alpha_i$ is a floating-point number, Summation computes $\gamma = \sum_i \alpha_i$. 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 suitable8 option to compute floating-point Summation is

---

8Kahan summation [36] incurs worst-case error of $2\varepsilon$ but requires $O(n)$ normalization and rounding steps, making it more expensive.
Algorithm FPSum\(^{p,q,n}\)(\(\{\alpha_i\}\_{i\in[n]}\))

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

Figure 1: Floating-Point Summation.

tree-sum (or pairwise summation), that recursively computes \(\gamma_n = \gamma_{2,1} + \gamma_{2,2}\), where \(\gamma_{2,1} = \sum_{i=1}^{2} \alpha_i\) and \(\gamma_{2,2} = \sum_{i=1}^{2} \alpha_{2+i}\). 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 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 \(e_{\text{max}}\) (Step 2). For \(\bar{n} = \log(n)\), define \(e_{\text{thr}} = e_{\text{max}} - \bar{n} - 1\) and threshold \(\Gamma = 2^{e_{\text{thr}}}\) such that only vector elements with magnitude \(\geq \Gamma\) contribute to the sum (Step 3).

Thus, we set the mantissa of all values with exponent \(< e_{\text{thr}}\) to 0 (Step 5).

2. For all \(i\), left-shift the mantissa of \(\alpha_i\) by \((\alpha_i.e - e_{\text{thr}})\), essentially setting the exponents of all elements to \(e_{\text{thr}}\) (Step 6). Note that a bitwidth of \(2q + \bar{n} + 2\) suffices for all left-shifted mantissas as \((\alpha_i.e - e_{\text{thr}}) \leq q + \bar{n} + 1\).

3. To be able to simply add the mantissas now, we convert the unsigned mantissas \(m_i^{(\text{align})}\) 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 \(\bar{n} + 1\) bits, i.e., needs to be done in \(\ell = 2q + 2\bar{n} + 3\) 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 \(\ell\)-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 \(e_{\text{thr}}\) has been carefully chosen to help obtain low numerical error in Theorem 1. This algorithm assumes a relation between \(p\) and \(n\). In particular, \(e_{\text{thr}}\) 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 \(\gamma\) and \(\gamma'\) represent the output of FPSum and result of summation over reals, respectively. Let \(\alpha_k\) be the element with the largest exponent. Note that \(|\alpha_k| \geq 2^{e_{\text{max}}}\) and up until the rounding step (before Step 12), FPSum computes the sum \(\gamma'\) of all elements \(\geq 2^{e_{\text{thr}}}\) exactly. Thus, the total magnitude of elements set to 0 in Step 5 that are ignored by FPSum is at most \(n \cdot 2^{e_{\text{thr}}} \leq 2^{e_{\text{max}}} \cdot 2^{2^{e_{\text{thr}}}} = |\alpha_k| \cdot 2^{-q-1}\). Hence, \(|\gamma - \gamma'| \leq |\alpha_k| \cdot \varepsilon\), where \(\varepsilon = 2^{-q-1}\) is the machine epsilon, and the relative error of \(\gamma'\) is \(|\Delta| = \frac{|\gamma - \gamma'|}{\gamma'} \leq \frac{|\alpha_k| \cdot \varepsilon}{\sum_{i \in [n]} |\alpha_i|} \leq \frac{\sum_{i \in [n]} |\alpha_i| \cdot |\alpha_i|}{\sum_{i \in [n]} |\alpha_i|} = \varepsilon \cdot \kappa\).
The final output $\hat{y}$ of FPSum is obtained after rounding $y'$ to $q$ mantissa bits. Thus, $\hat{y} = \gamma'(1 + \delta)$ [28], where $|\delta| \leq \epsilon$, and the absolute error of $\hat{y}$ w.r.t. $y$ is:

$$|y - \hat{y}| = |y - y'(1 + \delta)| = |y - y(1 + \Delta)(1 + \delta)| = |y(\delta + \Delta + \delta\Delta)| \leq |y| \cdot (\epsilon + \kappa + \epsilon^2\kappa).$$

Thus, the relative error of $\hat{y}$ is $\frac{|y - \hat{y}|}{|y|} \leq \epsilon (k + 1) + O(\epsilon^2\kappa)$.

**Secure Protocol.** Figure 2 describes the 2PC protocol corresponding to our Summation algorithm FPSum (Figure 1). For each step of the algorithm 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 $2^b$ mantissa $m_i$ by $r_i = (\alpha_i - e_{\text{thr}}) < (q + \bar{n} + 2)$. We store $r_i$ in $k = [\log(q + \bar{n} + 2)]$ 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 $\text{pow2}$ with $2^k$ entries over $\{0, 1\}^{q + \bar{n} + 2}$ such that $\text{pow2}[i] = 2^i$. We do a secret lookup in $\text{pow2}$ at index $r_i$ to learn shares of $2^i$. Then, we multiply with $m_i$ to obtain $m_i^{(\text{align})}$ in $2q + \bar{n} + 2$ bits. Next, to avoid overflows during addition of (aligned) mantissas and to accommodate the sign-bit, we require additional $\bar{n} + 1$ bits. Hence, in Step 10, we extend the mantissa to $\ell = 2q + 2\bar{n} + 3$ 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 $\Pi^{p + 2,n}_{\text{max}}$ (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 $60 \log n + 73$ rounds compared to $69 \log n$ in SECFLOAT that uses tree sum. Moreover, communication complexity of our protocol for FP32 and $n = 2000$ is $4.4 \times$ 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 $\alpha = (\alpha_1, \ldots, \alpha_n)$, where each $\alpha_i$ is an unnormalized floating-point value with a $(p + 2)$-bit signed exponent $\alpha_i, e \in [-2^{b - 1} + 1, 2^{b - 1})$ 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 $sc \geq q$.

Our algorithm for generalized Summation follows the blueprint of FPSum with the following modifications to deal with unnormalized mantissas. First, after computing $e_{\text{max}}$, we set $e_{\text{thr}} = e_{\text{max}} - sc - \bar{n} - (b - b')$ so that we can safely ignore the values smaller than the threshold $\Gamma = 2^{e_{\text{thr}}}$. With this change, the maximum shift amount is $sc + \bar{n} + (b - b')$ (as opposed to $q + \bar{n} + 1$) and consequently, we need to increase $\ell$ to $2b - b' + sc + 2\bar{n} + 1$ 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 $q - sc$ to the exponent.

We describe our algorithm for generalized Summation, g-FPSum, parameterized by $b, b', sc, p, q, n$ 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 $\Pi^{b, b', sc, p, q, n}_{\text{FPSum}}$ over secret shared input using the same steps as in Section 5.1.

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

**Proof.** Let $\alpha_b$ be the element with the largest exponent, and let $y$ and $y'$ represent the sum of all elements and sum of all elements with exponent $\geq e_{\text{thr}}$, respectively. We only need to argue that $|y - y'| \leq |\alpha_b| \cdot \delta$, where $\delta = 2^{-sc - 1}$, and the rest of proof follows in the same way as the proof of Theorem 1. It is easy to see that $|\alpha_b| \geq 2^{b - s + e_{\text{max}}}$ and each omitted term $\leq 2^{b - s + e_{\text{thr}} - 1}$. Thus, the sum of omitted terms has a magnitude $\leq n \cdot 2^{b - s + e_{\text{thr}} - 1} = 2^{b - s + e_{\text{max}} - sc - (b - b') - 1} = 2^{b - s + e_{\text{max}} - sc - 1} \leq |\alpha_b| \cdot 2^{2 - sc} - 1$ and $|y - y'| \leq |\alpha_b| \cdot \delta$. $

5.3 Dot product and matrix multiplication

Given floating-point vectors $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n)$, DotProduct is defined as $y = \sum_{i\in[n]} \alpha_i \cdot \beta_i$. Thus, dot product can be realized naively by first computing the intermediate products $\gamma = (\alpha_i \cdot \beta_i)_{i\in[n]}$ using floating-point multiplication protocol for scalars, $\Pi^{p, q}_{\text{FPMul}}$, from [60], and...
then calling our protocol for Summation, $\Pi_{\text{FPSum}}^p$ (Figure 2) on $\gamma$. 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 $\alpha_i$ and $\beta_j$ are added and the mantissas of $\alpha_i$ and $\beta_j$ are multiplied. The latter creates unnormalized $2q + 2$-bit mantissas with scale $2q$ and values 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 ($\Pi_{\text{FPSum}}^{p,q,n}$) to get the output of the dot product. Our protocol $\Pi_{\text{FPDotProd}}^p$ for dot product is formally described in Figure 7, Appendix E. For $n = 1000$, our approach has $1.2 \times$ lower communication than the naïve approach and is just as precise as proved below. The protocols for matrix multiplication and convolutions build on top of dot product, and their description is deferred to the full version of the paper.

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

*Proof.* We first look at the worst-case relative error of $\Pi_{\text{FPDotProd}}^p$. 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 $m_i^l$ as $m_i \in [2^{2q} \cdot (2^{q+1} - 1)^2]$, and thus, $m_i^l = \text{RN}(m_i, q) \in [2^q, 2^{q+2} - 2]$. Note that the rounding operation introduces a relative error of $|\delta_1| \leq \epsilon \cdot (2^{-q} - 1)$ to the intermediate product $\gamma$. Unlike $\gamma$, 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 $\Pi_{\text{FPSum}}^{p,q,n}$, which adds a relative error of $|\delta_2| \leq \epsilon(k + 1) + O(\epsilon^2 \kappa)$ to its output where $\epsilon = 2^{-q-1}$ (Theorem 2). Thus, the absolute error of $\Pi_{\text{FPDotProd}}^p$ is $|\sum (\gamma_i + \delta_2) - \alpha_i \beta_i| = |\sum \alpha_i \beta_i \cdot (1 + \delta_1) (1 + \delta_2) - \alpha_i \beta_i| \leq |\sum \alpha_i \beta_i \cdot (\epsilon(k + 1) + O(\epsilon^2 (k + 1)))|$, which implies a worst-case relative error of $\epsilon(k + 2) + O(\epsilon^2 (k + 1))$.

Now, we look at the worst-case error of the naïve solution. $\Pi_{\text{FPMul}}^p$ introduces a worst-case relative error of $\epsilon$ to the intermediate products, the same as $\Pi_{\text{FPDotProd}}^p$. 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 $\Pi_{\text{FPSum}}^{p,q,n}$ which has the worst-case relative error of $\epsilon(k + 1) + O(\epsilon^2 \kappa)$,
Invoke the dot-product protocol
Left-shift the mantissas of the BF16 input vectors by 1.

As discussed in Section 5, our protocols for linear layers build
6338    32nd USENIX Security Symposium
ation of BF16 that give performance improvements over
In this section, we discuss our techniques for secure imple-
q
of FP32 with the same dynamic range: mantissa bits
BFloat16 or BF16 is essentially a lower-precision version
Ππ
for the scale change, thereby obtaining an exact 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 follow-
generalized Summation would ignore more values than the
naïve approach (εthr depends on the scale sc and lower
scale leads to higher εthr 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 ΠΠ
. Our approach is much more ef-
ficient as it avoids the expensive steps of multiplication on
24-bit inputs (Step 3) and rounding by 23-bits (Step 6) in
ΠΠ
, as well as operates on mantissas of 16-bits as op-
posed to 25 bits in generalized summation. Our BF16 dot-
product protocol ΠΠ
 is described in Figure 4 and
its precision is proved below. From Section 5.3, we get the
BF16 matrix-multiplication protocol by building upon our
BF16 dot-product protocol.

Figure 3: Generalized Floating-Point Summation.

the same as ΠΠ
. Thus, ΠΠ
 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 imple-
mentation of BF16 that give performance improvements over
BEACON’s FP32 while being more precise than standard plat-
forms 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 straight-
forward manner to other representations, e.g., TensorFlow
which uses q = 10. Below, we discuss linear layers and defer
non-linear layers to the full version of the paper.

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 ΠΠ
 on
FP32 vectors.
3. Round the output mantissa obtained above by 16 bits using
ΠΠ
 to get the final BF16 output.

As can be seen easily, this protocol is at least as expensive as
ΠΠ
, 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 ΠΠ
 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 ΠΠ
, and has the same precision as the
standard BF16.

If we look closely at the naïve solution, it is first left-shift-
ing 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 mantis-
sas 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 follow-
ning generalized summation would ignore more values than the
naïve approach (εthr depends on the scale sc and lower
scale leads to higher εthr 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 ΠΠ
. Our approach is much more ef-
ficient as it avoids the expensive steps of multiplication on
24-bit inputs (Step 3) and rounding by 23-bits (Step 6) in
ΠΠ
, as well as operates on mantissas of 16-bits as op-
posed to 25 bits in generalized summation. Our BF16 dot-
product protocol ΠΠ
 is described in Figure 4 and
its precision is proved below. From Section 5.3, we get the
BF16 matrix-multiplication protocol by building upon our
BF16 dot-product protocol.

Theorem 4. The relative error of ΠΠ
 is at most
δκ + ε + O(εδκ), where ε = 2−8, δ = 2−24, and κ is the
condition number.

Proof. We first calculate the relative error of our protocol
ΠΠ
. We note that the intermediate products are
calculated exactly, unless they are much smaller than the
smallest representable exponent, i.e., αi,e + βi,e + 9 < −127.
Hence, the relative error of our scheme is same as relative
error introduced by generalized summation ΠΠ
, i.e.,
δκ + ε + O(εδκ) for ε = 2−8, δ = 2−24 by Theorem 2.

Corollary 1. ΠΠ
 is more precise than the naïve
solution that left-shifts input mantissas, performs ΠΠ
 and
rounds by ΠΠ
.
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 $6\times$ (Section 9.1), and achieves secure floating-point training with $<6\times$ 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 3.25 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 monochromatic 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 microbenchmarks 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

7 2PC for training and security proofs

We note that all our protocols for linear layers (Section 5,6) and non-linear layers (Appendix G) 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 the full version of the paper.

8 Implementation

We have implemented BEACON as a library in C++ on top of SECFLOAT with 5k LOC. This library’s API provides
SecFloat over BF16 tasks, BEACON improves latency by 6.3 – 16.1× and the communication by 5.4 – 9.5×. 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× lower latency and 6.2 – 6.5× 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). Sometimes KS22 goes out of memory 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× of KS22 for our training benchmarks. The communication of BEACON, on the other hand, is 21 – 100× 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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (minutes)</th>
<th>Comm. (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEACON</td>
<td>KS22</td>
</tr>
<tr>
<td>MNIST Logistic</td>
<td>56</td>
<td>16</td>
</tr>
<tr>
<td>MNIST FFNN</td>
<td>626</td>
<td>106</td>
</tr>
<tr>
<td>CIFAR LeNet</td>
<td>3,285</td>
<td>1,065</td>
</tr>
<tr>
<td>CIFAR HiNet</td>
<td>100,827</td>
<td>32,137</td>
</tr>
</tbody>
</table>

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

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 algorithms 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 multi-party 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


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 $100 \times 100$-sized matrix with another $100 \times 100$-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× 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× better than standard 32-bit floating-point.

E Dot Product Protocol $\Pi_{\text{FPDotProd}}^{p,q,n}$

Our protocol for dot product is formally described in Figure 7.
\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
 & \multicolumn{2}{|c|}{Time (s)} & \multicolumn{2}{|c|}{Comm (GiB)} \\
 & Linear & Non-Linear & Linear & Non-Linear \\
\hline
MNIST-Logistic & 44.2 & 1.3 & 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\% & 87877 & 122.8 & 99.86\% \\
\hline
\end{tabular}
\caption{Split in cost between linear and non-linear layers for SECFLOAT.}
\end{table}

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.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
 & Total & Norm. & % Norm. \\
\hline
Time (s) & 0.384 & 0.316 & 82.3\% \\
Comm (MiB) & 105.58 & 81.28 & 76.9\% \\
\hline
\end{tabular}
\caption{Split of SECFLOAT’s Addition. Norm. refers to normalization and rounding steps.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
 & Total & Summ. & % Summ. \\
\hline
Time (s) & 20.24 & 17.35 & 85.7\% \\
Comm (MiB) & 13668 & 10784 & 78.9\% \\
\hline
\end{tabular}
\caption{Split of SECFLOAT’s Matrix Multiplication. Summ. refers to Summation step.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
 & Microbenchmark & Time (s) & Comm. (GiB) \\
\hline
Summation-2k & FP32 & 10.29 & 9.05 \\
 & FP19 & 5.83 (1.77\times) & 4.9 (1.85\times) \\
 & BF16 & 5.18 (1.99\times) & 4.15 (2.18\times) \\
DotProduct-1k & FP32 & 4.65 & 4.17 \\
 & FP19 & 3.47 (1.34\times) & 2.94 (1.42\times) \\
 & BF16 & 2.74 (1.7\times) & 2.38 (1.75\times) \\
MatMul-100 & FP32 & 4.34 & 3.53 \\
 & FP19 & 2.94 (1.47\times) & 2.51 (1.41\times) \\
 & BF16 & 2.72 (1.59\times) & 2.1 (1.68\times) \\
\hline
\end{tabular}
\caption{Comparing cost of BEACON for various number representations. Improvement factor of FP19/BF16 over FP32 shown in parentheses.}
\end{table}

\section{F Helper Protocols}

\subsection{F.1 Normalization}

For a fixed-point value \(x \in \mathbb{Z}_2^s\) with scale \(s\), we use \([x]_{e,s}\) to denote the corresponding real value. Normalization takes an unnormalized mantissa \(m \in [Q] + 1\) 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 \(m' = m \ll (Q-k) \in [2^k,2^{Q+1})\), 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 \(Q-k\) 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.

\subsection{F.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 \(m \in [2^{Q+1} - 2^{Q+1}, 2^{Q+1})\) 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 \(m' = 2^q + 1\), \(e' = e + 1\) which doesn’t introduce any error and thus, the final output has relative error \(\leq e\).
Protocol $\Pi_{{\text{FP}_{\text{Dat Prod}}}}$  
**Input:** $i \in [n]$, $(\alpha, \beta)_{\text{FP}(p, q)}$, $(\gamma)_{\text{FP}(p, q)}$.  
**Output:** $\langle \gamma, \gamma' \rangle_{\text{FP}(p, q)}$ s.t. $\gamma = \sum_{i}^{n} \alpha \cdot \beta_{i}$.

1. for $i \in [n]$ do  
   2. Set $(\epsilon_{i})^{p+2} = (\alpha_{i}, e)^{p+2} + (\beta_{i}, e)^{p+2}$  
   3. Call $(m_{i})^{q+2} = \Pi_{\text{UMult}}(\alpha_{i}, m_{i}^{q+2} + \beta_{i}, m_{i}^{q+2})$  
   4. Set $(s_{i})^{B} = (\alpha_{i}, x)^{B} \oplus (\beta_{i}, x)^{B}$  
   5. Call $(z_{i})_{\text{FP}} = \Pi_{\text{QOR}}((\beta_{i}, z_{i})^{B})$  
   6. Call $(m_{i})^{q+2} = \Pi_{\text{QP}^{+}}(m_{i}, 2e^{2e+2})$  
   7. Call $(c)^{B} = \Pi_{\text{Mux}}((\epsilon_{i})^{p+2}, 1 - 2^{p-1})$  
   8. Call $(m'_{i})^{q+2} = \Pi_{\text{Mux}}((c)^{B}, 0, (m_{i})^{q+2})$  
   9. Call $(\epsilon'_{i})^{p+2} = \Pi_{\text{Mux}}((c)^{B}, 1 - 2^{p-1}, (\epsilon_{i})^{p+2})$  
10. Set $(\delta_{i})_{\text{FP}} = \langle (x_{i})^{B}, (s_{i})^{B}, (\epsilon'_{i})^{p+2}, (m_{i})^{q+2} \rangle_{\text{FP}^{q+2}}$  
11. Return $\langle \gamma, \gamma' \rangle_{\text{FP}(p, q)} = \Pi_{\text{FPSum}}(\{\delta_{i}\}_{\text{FP}})$

Figure 7: Protocol for Dot Product.

Protocol $\Pi_{{\text{FP}_{\text{Normilize}}}}$  
**Input:** Unnormalized $(m_{i})^{q+1}, (e)^{p+2}$, scale of $m$ is $q$.  
**Output:** Normalized $(m'_{i})^{q+1} = (e)^{p+2}$.

1. Call $(k)^{Q+1}, (K)^{Q+1} = \Pi_{\text{TLSN}}((m_{i})^{Q+1})$  
2. Call $(m'_{i})^{Q+1} = \Pi_{\text{UMult}} ((m_{i})^{Q+1}, (K)^{Q+1})$  
3. Set $(e')^{p+2} = (\epsilon)^{p+2} + \langle (k)^{Q+1} \text{ mod } 2^{p+2} \rangle^{q} - q$  
4. Return $(m')^{Q+1}, (e')^{p+2}$

Figure 8: Normalize mantissa and adjust exponent accordingly.

F.3 Clip details

For target representation $(p, q)$ and input $\alpha$, the clipping protocol $\Pi_{\text{Clip}}$ is easily realized by first performing a comparison $(c)^{B} = \Pi_{\text{Lt}}(\langle c, e \rangle^{p+2}, 2^{p-1} + 2)$ 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: $\Pi_{\text{Mux}}(\langle c, e \rangle^{B}, 0, (\alpha, m)^{q+1})$ and $\Pi_{\text{Mux}}(\langle c, e \rangle^{B}, -2^{p-1} + 1, (\alpha, e)^{p+2})$, respectively. Since the comparison operation is signed, we require $|\alpha, e| < 2^{p+1}$ for its correctness.

G 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.

G.1 ReLU

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

G.2 Softmax

Given vector $\alpha \in \mathbb{R}^{n}$ as input, softmax outputs a vector $\delta$ such that $\delta_{i} = \exp(\alpha_{i}) / \sum_{j} \exp(\alpha_{j})$. In practice to avoid overflows in exponentiation, the maximum element is subtracted from every element of the array to create $\beta$, i.e., $\beta_{i} = \alpha_{i} - \alpha'$, where $\alpha' = \max_{j} \alpha_{j}$. At a high level, softmax has the following steps:

1. Compute maximum element $\alpha'$ and subtract it from every vector element to get the vector $\beta$ (with all negative entries).

2. Compute exponentiation on $\beta$ to get $\gamma$.

3. Sum the elements of $\gamma$ to get $\theta$.

4. Divide $\gamma$ by $\theta$ 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 $\theta$ 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.

G.3 Sigmoid/Tanh

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