SECFLOAT: Accurate Floating-Point meets Secure 2-Party Computation

Deevashwer Rathee
UC Berkeley
deevashwer@berkeley.edu

Anwesh Bhattacharya
Microsoft Research
t-anweshb@microsoft.com

Rahul Sharma
Microsoft Research
rahsha@microsoft.com

Divya Gupta
Microsoft Research
divya.gupta@microsoft.com

Nishanth Chandran
Microsoft Research
nichandr@microsoft.com

Aseem Rastogi
Microsoft Research
aseemr@microsoft.com

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833697

***Abstract*—We build a library SECFLOAT for secure 2-party computation (2PC) of 32-bit single-precision floating-point operations and math functions. The existing functionalities used in cryptographic works are imprecise and the precise functionalities used in standard libraries are not crypto-friendly, i.e., they use operations that are cheap on CPUs but have exorbitant cost in 2PC. SecFloat bridges this gap with its novel crypto-friendly precise functionalities. Compared to the prior cryptographic libraries, SECFLOAT is up to six orders of magnitude more precise and up to two orders of magnitude more efficient. Furthermore, against a precise 2PC baseline, SECFLOAT is three orders of magnitude more efficient. The high precision of SECFLOAT leads to the first accurate implementation of secure inference. All prior works on secure inference of deep neural networks rely on ad hoc float-to-fixed converters. We evaluate a model where the fixed-point approximations used in privacy-preserving machine learning completely fail and floating-point is necessary. Thus, emphasizing the need for libraries like SECFLOAT.**

I. INTRODUCTION

Floating-point is the default format used to perform operations on real numbers on modern computer hardware. All CPUs support efficient and accurate floating-point arithmetic on which software developers rely upon. Whether it is solving differential equations, continued fractions, matrix factorization, or financial predictions, floating-point is everywhere. Theoretically, it was shown in the 1980's [34], [73] that any program, and therefore, floating-point programs, can be securely implemented using 2-party computation (2PC) protocols. These protocols allow two mutually distrustful parties to run programs on their secret inputs in a way that both the parties learn nothing about the inputs of each other beyond what is revealed by the outputs (Section IV-C). However, since almost all practical software uses floating-point arithmetic and existing library support for 2PC of floating-point is either non-existent or imprecise, there is a big gap between the software that exists and the programs that can actually be run accurately with existing 2PC implementations.

When developers use (single-precision) floating-point math libraries like GNU's libm or Intel's MKL [6], [7], they get strong accuracy guarantees. In particular, the results of *primitive operations* like floating-point addition and division are *correct* (return the floating-point number closest to the ideal real result, Section III-B) and the implementations of *math functions* in `math.h` like trigonometric sine are *precise* (return either the closest or the second-closest floating-point number to the ideal real result, Section III-C). With these guarantees, the developers can build applications that represent computations over reals accurately while using only 32-bits.

A. Secure computation of floating-point code

Running floating-point code with 2PC while maintaining accuracy is hard. Although efficient 2PC support for integer arithmetic exists, it is insufficient for floating-point. Single-precision floating-point numbers or *floats* differ from integers as, even when using only 32-bits, floats have a large *dynamic range*: they can represent very large numbers like 2^{127} and very small numbers like 2^{-126} exactly. As 2PC protocols can support arbitrary bitwidth integers, it is theoretically feasible to exactly simulate operations on floats with operations on integers or fixed-point numbers. However, due to the large range of values that floats represent, this approach requires very high bitwidth (over 512-bit) integers that impose intractable performance overheads.

The primary work on 2PC of floating-point is by Demmler et al. [27] (referred to as ABY-F to stand for ABY-float). They express floating-point operations as optimized circuits (obtained from hardware synthesis tools) and run them with generic 2PC protocols[1]. In contrast, MP-SPDZ [42] uses hand-made custom functionalities for floating-point operations given by Aliasgari et al. [10], [11] (referred to as SA-F). The users of existing cryptographic implementations of floating-point (ABY-F and SA-F) face two challenges. First, the available math implementations are *imprecise* and don't provide the accuracy guarantees that are provided by standard implementations of `math.h` (Section III-C, Section IX-A). In particular, we have observed that ABY-F and SA-F produce garbage output (floating-point exceptions like division-by-zero or answers that are wrong in every digit) for valid computations (Appendix D). We show two very simple examples of these errors. First, when evaluating $\frac{x}{\sin\pi(x)}$ at x near zero, the

[1]EMP [2] also uses the circuits of ABY-F. We discuss more in Section IX.

DOI 10.1109/SP46214.2022.00124

expected output is $0.3 \approx \frac{1}{\pi}$ but ABY-F circuits do a division by zero. Second, when evaluating the derivative of $\log(x)$ at $x = 1$, i.e., $\frac{d\log(x)}{dx}|_{x=1} \approx \frac{\log(1)-\log(1-\epsilon)}{\epsilon}$, the expected output is 1.0 but ABY-F outputs 0.7. Hence, the applicability of ABY-F to safety critical applications is unclear (SA-F is even more imprecise than ABY-F). Second, even if the target computation is insensitive to precision, both ABY-F and SA-F suffer from high performance overheads (Section IX-A).

B. Our contributions

We provide SECFLOAT, the first secure 2PC library for floating-point that matches the accuracy of Intel's libraries. SECFLOAT supports both *correct* floating-point primitive operations (addition, multiplication, division, and comparison) and *precise* math functions (trigonometric functions, exponentiation, and logarithm). In particular, for the examples above, SECFLOAT produces the expected outputs. Furthermore, SECFLOAT outperforms a state-of-the-art precise baseline [39] by three orders of magnitude (Section IX-A) and is up to two orders of magnitude more efficient than even imprecise baselines [2], [27], [42]. Because of SECFLOAT's better performance, even for the computations that are less sensitive to precision, users of secure 2PC of floating-point are better-off using SECFLOAT compared to the existing systems.

We also implement privacy preserving proximity testing [70], an application that has previously been used to evaluate 2PC implementations of floats [27]. On random input coordinates from Google maps, SECFLOAT-based implementations are four orders of magnitude more accurate on average and have $6.8 - 11\times$ lower communication than the state-of-the-art. Here, the average error in distance computed by Intel's MKL/SECFLOAT from ideal real result is about a millimeter and it is about two metres for ABY-F ($1600\times$ worse).

Finally, prior works in secure inference use fixed-point arithmetic. We show a case study in privacy-preserving advertisements where fixed-point models completely fail and floating-point is necessary (Section IX-B).

The main technical contributions of SECFLOAT are novel functionalities for operations over floats that a) keep the cryptographic cost of running the functionalities low and b) provide accurate results. To obtain 2PC protocols for these floating-point functionalities, we run each atomic computation with a 2PC building block over integers (Table I), e.g., comparison, multiplication, etc. We need to design new functionalities because the existing functionalities used in cryptographic works are imprecise and the precise functionalities used in non-cryptographic works are not *crypto-friendly*, i.e., they use operations that are cheap on CPUs but have exorbitant cost in 2PC. SECFLOAT bridges this gap with its crypto-friendly precise functionalities. Note that although each building block of Table I corresponds to a single x86 assembly instruction, their communication costs (given in the last column), which are the bulk of the cryptographic costs, are very different.

Next we discuss the standard libraries, i.e., the non-cryptographic floating-point resources, and why they are unsuitable for 2PC. Then, in Section I-D we provide an overview of SECFLOAT's functionalities.

C. Why standard libraries are not suitable for 2PC?

1) IEEE Standard: The IEEE standard defines the representation for *floats* with $p = 8$ exponent bits and $q = 23$ mantissa bits [3] and specifies that primitive operations should be *correct*. That is, the primitive operations on these numbers should be computed with infinite precision real arithmetic and then rounded to the floating-point number closest to the ideal real result. The IEEE standard leaves it to the hardware designers to come up with efficient functionalities that compute intermediate results equivalent to the infinite precision results. The existing implementations of IEEE standard are not crypto-friendly and lead to inefficient 2PC implementations (e.g., they use Boolean circuits [71] or high bitwidth integers [37]).

2) Intel's MKL: Although IEEE does not specify the precision of math libraries, it is standard to compute *precise* results. For math functions, there is a standard recipe [25] with three steps which is used by all standard math libraries including MKL: *range reduction*, *polynomial approximations*, and *output compensation*. For example, in the first step, computing a math function $z = 2^x$ on a large input range like $x \in [-2^{126}, 2^{127}]$ is reduced to computing 2^f on a very small input range $f \in [0, 1)$ using high school mathematics ($2^x = 2^N 2^f$ where N is an integer and $0 \leq f < 1$). The second step uses polynomials to compute $y \approx 2^f$ accurately enough so that when y is mapped to the final float output ($z = y \cdot 2^N$) in the third step, the output is *precise*. Standard libraries use high bitwidths to get *precise* results that have exorbitant cost when implemented as-is using 2PC. Below, we discuss the 2PC performance bottlenecks in more detail.

D. SECFLOAT's design

SECFLOAT has novel and accurate crypto-friendly functionalities. Standard floating-point libraries incur high worst case execution time (WCET) on corner cases to run likely inputs efficiently. However, in 2PC, the floating-point implementations must take the same time on all inputs to ensure security. Hence, SECFLOAT's functionalities require low WCET, which is one of the reasons why the standard libraries are unsuitable for it. Furthermore, the existing custom functionalities that have been designed for MPC are imprecise [10], [11].

For primitive operations over floats, i.e., addition, multiplication, and division, SECFLOAT meets the specifications of IEEE standard and more. In particular, the primitive operations of SECFLOAT are parametrized by p and q and provide *provably* correct results for all floating-point representations, including the ones that are outside the IEEE standard, e.g., Google's BFloat16 with $p = 8$ and $q = 7$, nVIDIA's TensorFloat32 with $p = 8$ and $q = 10$, etc. This generality of primitive operations is crucial for SECFLOAT's math functionalities. Furthermore, they use algorithms that are crypto-friendly. For example, division looks up the approximate result from a moderately sized lookup table which is further refined to a correct result using Newton iterations. This scheme is

suboptimal for CPUs but outperforms the state-of-the-art in 2PC by $11.6\times$. Standard implementations of the IEEE standard are overfitted to $q = 23$ and lead to $2.36 - 6.07\times$ slower 2PC implementations than SECFLOAT (Section IX-A3). For this evaluation, we simplified implementations in Berkeley's SoftFloat [37] by removing the code for special values like NaNs and denormals, and executed these simplified functionalities with the same cryptographic building blocks as SECFLOAT. The former is necessary for a fair comparison as SECFLOAT, like the other baselines, doesn't support NaNs and denormals (Section III-A). Adding support for denormals and NaNs would require using comparisons on inputs to identify whether special values can arise in the computation and multiplexing the results of normal computations and computations with special values. Here, the cost of normal computations is much higher than the other costs.

Design of math functions. For math functions, SECFLOAT also follows the standard 3-step recipe (Section I-C2). However, the similarities between SECFLOAT and MKL end here. Since 2PC, which is the target for SECFLOAT, has very different performance characteristics than CPUs, SECFLOAT's functionalities are very different from MKL in the representations, range reductions, and polynomial approximations. We make four technical contributions to design cryptographically efficient math functionalities.

First, we provide new range reductions and implement them efficiently with 2PC. Although there are several range reduction techniques available in literature [25], [49], [51], they rely on polynomial approximations to be evaluated with very high precision, e.g., doubles with $q = 52$, which has high cryptographic cost. Hence, our range reductions are more complex but allow the polynomial approximations to use $q = 27$ and still permit the overall functionalities to be precise. Note that evaluating polynomials over custom representations with $q = 27$ is made possible by the generality of SECFLOAT's primitive operations.

Second, MKL uses high degree polynomials that are not crypto-friendly. We show how to obtain crypto-friendly splines (Appendix K). Third, we provide a new general functionality to securely evaluate splines efficiently that can have applicability beyond floating-point arithmetic (see Section VI-A).

Fourth, to further improve performance, we use cheaper floating-point primitive operators whenever possible that have lower cost but don't provide *correct* results for all inputs. SECFLOAT ensures that the cumulative error (the sum of approximation errors from splines, rounding errors when computing the polynomial with cheaper floating-point operations, and the rounding errors arising from output compensation) remains low enough that the overall functionalities are *precise*.

In Section II, we provide a walk-through example to further elaborate on SECFLOAT's design choices.

Bitwidth optimizations. The functionalities of primitive operations in SECFLOAT use operations over integers or fixed-point numbers that can be realized using 2PC frameworks such as ABY [28], SPDZ [42], etc. These frameworks use a uniform bitwidth, e.g., 64, for all values. This restriction of uniform bitwidth mandates use of larger than necessary bitwidths for many sub-computations that is wasteful for SECFLOAT's functionalities and leads to unnecessary performance penalties. As an additional optimization, SECFLOAT's functionalities use mixed-bitwidth computation and ensure that expensive high bitwidths are used only sparingly: the computations use as low bitwidths as possible, switch to high bitwidths via extensions when necessary, and then come back to low bitwidths via truncations. Some of the building blocks used by our functionalities are sign or zero extension of an n_1-bit integer to an n_2-bit integer ($n_1 < n_2$), truncating an n_1-bit integer to an n_2-bit integer ($n_1 > n_2$), multiplying an n_1-bit integer and an n_2-bit integer into an $n_1 + n_2$-bit result, etc (Section IV-D). Note that these building blocks are generic and not specific to floating-point. Furthermore, SECFLOAT's functionalities of math functions use operations over both floating-point numbers and integers. Integer operations in SECFLOAT's math functions also use minimal possible (necessary to maintain precision) bitwidths and rely on the same mixed-bitwidth building blocks discussed above. We use the state-of-the-art 2PC protocols for the mixed-bitwidth building blocks provided in SIRNN [63]. More efficient 2PC protocols for these building blocks would improve the performance of SECFLOAT's functionalities even further.

To summarize, we make the following contributions:

- We provide the first crypto-friendly functionalities for precise math functions. These use new range reductions, a new functionality to evaluate splines, and a mechanism to generate splines that are efficient to evaluate in 2PC (Section VI and Appendix K).
- We provide crypto-friendly functionalities for primitive floating-point operations that are correct and support arbitrary floating-point representations (Section V).
- We build a library SECFLOAT[2] for secure 2-party computation of 32-bit floating-point arithmetic which is up to six orders of magnitude more accurate and up to three orders of magnitude more efficient than prior work (Section IX).
- Using privacy-preserving proximity testing, we show that SECFLOAT outperforms prior floating-point 2PC libraries in *both* accuracy and performance. We provide an example of machine learning inference with 2PC that concretely justifies the use of floating-point over fixed-point arithmetic (Section IX-B).

E. Other Related Work

There is prior work that focuses on implementing floating-point in the honest majority setting [21], [22], [41], [43], [46], [55] but these custom functionalities are imprecise [61]. Moreover, we focus on the 2PC setting that provides stronger security guarantees. It is possible to use CBMC-GC to get correct functionalities for primitive operations [13], [61].

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

However, ABY-F [27] outperforms it on primitive operations and the overheads of this approach are intractable for math functions (Section IX-A3). *Fixed-point* math implementations include [12], [20], [24], [38], [44], [45], [53], [63], [66]. Golden section numbers [30] is an alternative real number representation that has been evaluated in the honest-majority setting. HE-based works [14], [15], [54], [59], [74] focus on floating-point addition/multiplication and are less performant than ABY-F. Orthogonal to SECFLOAT is the work on OTs with sublinear communication [18], [26], [72] that when combined with SECFLOAT can further improve the performance in low-bandwidth settings [63] (Appendix F).

II. WALK-THROUGH EXAMPLE

To explain the differences between standard math libraries and SECFLOAT, we provide an example where we start with a standard implementation $\mathcal{L}$ of $\log_2 x$ and incrementally modify it to obtain SECFLOAT's crypto-friendly functionality. First, $\mathcal{L}$ expresses the input float x as $x = m2^N$ with $m = 1 + \delta$ where δ is a double s.t. $0 \leq \delta < 1$, and N is an integer. Using the range reduction $\log_2 x = N + \log_2(1 + \delta)$, $\mathcal{L}$ first computes $y \approx \log_2(1 + \delta)$ using a polynomial approximation which uses the Taylor series polynomial with degree seven: $y = \delta - \frac{\delta^2}{2} + \ldots - \frac{\delta^7}{7}$. Here, all the arithmetic operations are performed using doubles. Then, $\mathcal{L}$ rounds the double $y + N$ to a float f and returns f. Although f is *precise*, $\mathcal{L}$ is not crypto-friendly because it uses high degree polynomials over high bitwidth doubles.

To make $\mathcal{L}$ crypto-friendly, we first replace doubles having $q = 52$ mantissa bits with a representation having $q = 27$. Each floating-point operation can introduce a rounding error of $\frac{1}{2^q}$ [33]. With $q = 27$, the computation $y + N$ suffers from high numerical errors. When $N = -1$ and $\delta \approx 1$ then $y \approx 1$ and subtracting *approximately* computed values that are close to each other leads to cancellation errors (here, y is approximate because of the rounding errors caused in the polynomial evaluation). These errors were insignificant when y was a double with $q = 52$ but they tank the precision with $q = 27$. Hence, SECFLOAT uses the following novel range reduction: when $N \neq -1$ then $\log_2 x = N + \log_2(1 + \delta)$ (as before) and when $N = -1$ then $\log_2 x = \log_2(1 - \delta')$, where $\delta' = \left(1 - \frac{m}{2}\right)$. Note that the value of δ' computed using finite-bit arithmetic over m is provably exactly equal to the value obtained if δ' were to be computed with operations performed over infinite-bit real numbers using standard results from numerical analysis (e.g., Sterbenz's theorem [69]).Thus, SECFLOAT's range reduction avoids cancellation errors.

Next, we replace high degree polynomials with low degree (degree 3) piecewise polynomials or *splines* that are more efficient in 2PC. Hence, $\log(1+\delta) \approx \theta_0^{(i)} + \theta_1^{(i)}\delta + \theta_2^{(i)}\delta^2 + \theta_3^{(i)}\delta^3$ if $\delta \in [\kappa^{(i)}, \kappa^{(i+1)})$. We show how to compute $\theta_j^{(i)}$s and $\kappa^{(i)}$s in Appendix K. To efficiently compute this step, we provide a new functionality that given a public spline and a secret δ finds $(\theta_0^{(i)}, \theta_1^{(i)}, \theta_2^{(i)}, \theta_3^{(i)})$ s.t. $\delta \in [\kappa^{(i)}, \kappa^{(i+1)})$ (Section VI-A).

Finally, we replace the $q = 27$ additions in spline evaluation with cheaper additions that have lower cryptographic cost at the expense of larger errors than the correct additions (multiplications are still correct). We then recompute the splines to ensure that the new splines lead to math functionalities with provably *precise* outputs while using the cheap incorrect additions. We show the final SECFLOAT functionality for $\log_2 x$ in Figure 8 that has been exhaustively tested for *precise* results on all possible floating-point inputs (Section VI).

This example demonstrates that SECFLOAT strikes a delicate balance between cryptographic costs and numerical errors, which has never been achieved before: prior cryptographic libraries are imprecise and the standard math libraries are inefficient with 2PC (Section IX).

III. FLOATING-POINT BACKGROUND

Floats in the IEEE standard [3] consist of a sign bit s, an 8-bit unsigned integer η called the biased exponent, and 23 mantissa bits ω. These encode the real number $(-1)^s \cdot 2^{\eta-127} \cdot 1.\omega$. We use the standard notation that binary strings are prefixed with `0b`. For example, $s = 0, \eta = 127, \omega = \texttt{0b0}^{22}\texttt{1}$ (a 23-bit binary string with top 22 bits zero and least significant bit 1), represents the real number $1 + 2^{-23}$. This representation doesn't enjoy nice algebraic properties that are critical for cryptography. For example, the numbers are not evenly distributed which makes masking difficult. Hence, performing secure computation over floating-point is usually more expensive than cryptography over integers [27].

A. Special values

Our aim with SECFLOAT is to provide the same floating-point support that standard software operates with. IEEE standard defines four special values: $\pm$`inf` (the largest and the smallest representable values), ± 0, NaNs (that are result of $0/0$, $\log(-1)$, etc.) and denormals (small numbers to enable gradual underflows). The standard libraries for floating-point software don't handle NaNs and denormal special values correctly by default. For example, Intel's math libraries, with optimizations turned on, doesn't handle denormals correctly and requires extra compilation flags to enable strict adherence to the IEEE standard. The reason for this design is two fold: handling these special values correctly incurs a significant performance cost and writing software that uses these special values properly requires specialized expertise in numerical analysis which is beyond the scope of most programmers. The algebra of these special values is counter-intuitive, e.g., $x = x$ is false when x is NaN. Hence, although there is no conceptual difficulty in supporting denormals, NaNs, and floating-point exceptions, neither SECFLOAT nor the baselines we compare with (Section IX) are IEEE compliant for them.

B. Correct results

The IEEE standard requires that the result of primitive operators, i.e., addition, subtraction, multiplication and division, is *correctly rounded*. That is, the result must be computed exactly (with infinite precision) and then rounded to the floating-point number nearest to the exact result (using round to even in case of ties). To perform these operations, the intermediate

computations require more bits. For example, it is common for handheld calculators to internally work with 13 decimal digits so that the user gets precise operations on 10 digits. We call the correctly rounded results as *correct* results as they are the most precise results permitted by the representation.

We provide some examples of the default IEEE rounding mode using a representation with two mantissa bits, i.e., $|\omega| = 2$. The binary string $\texttt{0b}d_j \dots d_0.d_{-1} \dots d_{-k}$ represents the real number $\sum_{i=-k}^{j} d_i 2^i$. Here, with $|\omega| = 2$, the representable numbers are $\{\dots, \texttt{0b1.01} = 1.25, \texttt{0b1.10} = 1.5, \texttt{0b1.11} = 1.75, \dots\}$. The other real numbers are not exactly representable as floating-point numbers and need to be rounded to floating-point numbers. In IEEE, the real number $1.3125 = \texttt{0b1.0101}$ would be rounded down to $1.25 = \texttt{0b1.01}$. In contrast, $1.4375 = \texttt{0b1.0111}$ would be rounded up to $1.5 = \texttt{0b1.10}$. Now, consider $1.375 = \texttt{0b1.0110}$ which is half way between 1.25 and 1.5. It is rounded up to 1.5 as IEEE rounds to even in case of ties. Finally, $1.625 = \texttt{0b1.101}$, which is half way between 1.5 and 1.75 is rounded down to 1.5 because of round to even. Formally, in case of ties, the real number is rounded to the floating-point number whose least significant mantissa bit is zero. Although this rounding scheme is more complicated than always rounding down or rounding up, Reiser and Knuth [65] provide compelling arguments about how its use helps maintain the precision in numerical software. Hence, round to nearest with ties to even is the default rounding mode in IEEE standard and also the one that SECFLOAT uses.

Next, lets consider an example of an operation on a floating-point number $x = 1.25 = \texttt{0b1.01} = (-1)^0 2^{127-127} 1.01$ with $\omega = \texttt{0b01}$ and $\eta = 127$. Then $x^2 = \texttt{0b1.01} * \texttt{0b1.01} = \text{round}(1.25 \cdot 1.25) = \text{round}(1.5625) = \text{round}(\texttt{0b1.1001}) = \texttt{0b1.10} = 1.5$. Here, first the intermediate result is computed with high bitwidth to get the exact real result 1.5625 which is then subsequently rounded to only two mantissa bits to obtain the final result 1.5 with $\omega' = \texttt{0b10}$ and $\eta' = 127$. For primitive operations, some extra bits in intermediate values is sufficient to get the exact real result [3]. However, this is not the case with math functions.

C. Precise results

The IEEE standard does not require the implementations of math functions, i.e., transcendental functions to be correctly rounded because of the *table maker's dilemma* [33]: computing the correctly rounded outputs of transcendental functions for $|\omega|$ bits of mantissa (say 23 bits) can require intermediate values with unbounded bits of mantissa. However, most math library implementations try to ensure that the computed result is "close" to the exact real result. This closeness relation is formally defined using an error metric[3] known as ULPs ("units in last place") [33]. In particular, correctly rounded results have ULP error ≤ 0.5 and commonly used implementations of math libraries have ULP error ≤ 1.

Formally, the ULP error between a floating-point number r' and a real number r is defined as follows. We define $\text{ulp}(r)$ as the distance between the two floating-point numbers that surround r. Hence, if r^- is the floating-point number just below r and r^+ is the floating-point number just above r then $\text{ulp(r)} = (r^+ - r^-)$. Then the ULP error between r and r' is $\text{ULP}(r, r') = \frac{|r - r'|}{\text{ulp}(r)}$. For IEEE-compliant primitive operations, the ULP error is at most half and this upper bound is reached when $|r^- - r| = |r^+ - r|$. Most commonly used math libraries like Intel's MKL guarantee that if the exact real output of a transcendental function is r then the output r' is either r^- or r^+, which ensures that the ULP error between r' and r is below one. For *imprecise* implementations with $u > 1$ ULP error, $\log_2\ u$ bits of the output are *garbage* [33].

[3]Relative errors are undefined at zero and deducible from ULP errors [33]. In contrast, ULP errors are always well defined. Absolute errors are less useful, e.g., in exponentiation, the maximum absolute error of MKL/SECFLOAT/all baselines exceeds 10^{38}.

IV. PRELIMINARIES

We define notation, secret sharing, background on 2PC with its security model, and 2PC building blocks.

A. Notation

Let λ be the computational security parameter. $[k]$ refers to the set $\{0, \dots, k-1\}$, $x||y$ denote concatenation of strings x and y, and $\mathbf{1}\{P\}$ be the indicator function that returns 1 if the predicate P is true and 0 otherwise. We use the natural one-to-one mapping between $\{0,1\}^\ell$ (ℓ-bit integers) and the ring $\mathbb{Z}_{2^\ell}$. For an element $x \in \mathbb{Z}_{2^\ell}$, we refer to its unsigned representation as $\mathsf{uint}_\ell(x) = \zeta_\ell(x)$, where ζ_ℓ is a lossless lifting operator that maps an element of $\mathbb{Z}_{2^\ell}$ to an element in $\mathbb{Z}$. We use 2's complement encoding for signed representation, which is defined as $\mathsf{int}_\ell(x) = \mathsf{uint}_\ell(x) - \mathsf{MSB}(x) \cdot 2^\ell$, where $\mathsf{MSB}(x)$ is the most-significant bit of x.

Fixed-point representation. Real numbers can be encoded as integers using *fixed-point* representation parameterized by bitlength ℓ and scale s. For an unsigned fixed-point integer x, $[\![x]\!]_{\ell,s}$ denotes its real value $\frac{\mathsf{uint}_\ell(x)}{2^s} \in \mathbb{Q}$.

B. Secret sharing

We use 2-out-of-2 additive secret sharing schemes over different rings [16], [67] between two parties P_0 and P_1. An ℓ-bit secret-shared integer is represented as $\langle x \rangle^\ell = (\langle x \rangle_0^\ell, \langle x \rangle_1^\ell)$ with $x = \langle x \rangle_0^\ell + \langle x \rangle_1^\ell \bmod 2^\ell$ and share of P_b is $\langle x \rangle_b^\ell$ for $b \in \{0,1\}$. To denote shares of boolean values over $\mathbb{Z}_2$, we use the superscript B. 2-out-of-2 secret sharing schemes guarantee that each share contains no information about the secret (i.e., the distribution of $\langle x \rangle_b^\ell$ for $b \in \{0,1\}$, is independent of x).

C. 2PC and Security Model

Secure 2-party computation (2PC) [34], [73] considers two parties P_0 and P_1 with private inputs x and y respectively, who agree to compute a public function f on these inputs, i.e., $f(x, y)$. 2PC provides an interactive protocol with the guarantee that no party learns anything more than the output of the function. A common technique for constructing 2PC protocols is to convert computation over cleartext values into

Functionality	Notation		Description	Communication
	Functionality	Protocol		
Multiplexer [64]	$z = c \ ? \ x : y$	$\langle z\rangle^\ell = \Pi^\ell_{\mathsf{MUX}}(\langle c\rangle^B, \langle x\rangle^\ell, \langle y\rangle^\ell)$	$z = x$ if $c = 1$, else $z = y$	$2\lambda + 2\ell$
OR [64]	$z = x \vee y$	$\langle z\rangle^B = \Pi_{\mathsf{OR}}(\langle x\rangle^B, \langle y\rangle^B)$	$z = x \vee y$	$\lambda + 20$
Equality [64]	$e = \mathbf{1}\{x = y\}$	$\langle e\rangle^B = \Pi^\ell_{\mathsf{EQ}}(\langle x\rangle^\ell, \langle y\rangle^\ell)$	Checks if $x = y,\ x, y \in \mathbb{Z}_{2^\ell}$	$< \frac{3}{4}\lambda\ell + 9\ell$
Comparison [64]	$c = \mathbf{1}\{x > y\}$	$\langle c\rangle^B = \Pi^\ell_{\mathsf{GT}}(\langle x\rangle^\ell, \langle y\rangle^\ell)$	Checks if $x > y,\ x, y \in \mathbb{Z}_{2^\ell}$	$< \lambda\ell + 14\ell$
Lookup Table (LUT) [29]	$y = L(x), y \in \mathbb{Z}_{2^n}$	$\langle y\rangle^n = \Pi^{m,n}_{\mathsf{LUT}}(L, \langle x\rangle^m)$	index x, LUT L, $z \in \mathbb{Z}_{2^n}$	$2\lambda + 2^m n$
Zero-Extension [63]	$y = \mathsf{ZXt}(x, n)$	$\langle y\rangle^n = \Pi^{m,n}_{\mathsf{ZXt}}(\langle x\rangle^m)$	$\zeta_n(y) = \zeta_m(x) \bmod 2^n, m \le n$	$\lambda(m+1) + 13m + n$
Truncate-and-Reduce [63]	$y = \mathsf{TR}(x, s)$	$\langle y\rangle^{\ell-s} = \Pi^{\ell,s}_{\mathsf{TR}}(\langle x\rangle^\ell)$	Upper $\ell - s$ bits of x	$\lambda(s+1) + \ell + 13s$
Unsigned Mixed-bitwidth Multiplication [63]	$z = x *_\ell y$	$\langle z\rangle^\ell = \Pi^{m,n,\ell}_{\mathsf{UMult}}(\langle x\rangle^m, \langle y\rangle^n)$	$\zeta_\ell(z) = \zeta_m(x) \cdot \zeta_n(y) \bmod 2^\ell$, $\ell \ge \mathsf{max}(m, n)$	$\lambda(3\mu + \nu) + \mu(\mu + 2\nu)$ $+16(m + n)$
Signed Mixed-bitwidth Multiplication [63]	$z = x *'_\ell y$	$\langle z\rangle^\ell = \Pi^{m,n,\ell}_{\mathsf{SMult}}(\langle x\rangle^m, \langle y\rangle^n)$	$\mathsf{int}_\ell(z) = \mathsf{int}_m(x) \cdot \mathsf{int}_n(y) \bmod 2^\ell$, $\ell \ge \mathsf{max}(m, n)$	$\lambda(3\mu + \nu) + \mu(\mu + 2\nu)$ $+16(m + n)$
Most Significant Non-Zero Bit [63], [73]	$k, K = \mathsf{MSNZB}(x)$	$\langle k\rangle^\ell, \langle K\rangle^\ell = \Pi^\ell_{\mathsf{MSNZB}}(\langle x\rangle^\ell)$	k, s.t. $x_k = 1 \wedge \forall i > k, x_i = 0$, $K = 2^{\ell-1-k}$	$\le \lambda(5\ell - 4) + \ell^2$

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

corresponding cryptographically secure interactive protocols that operate over secret shares. For instance, say P_0 and P_1 wish to compute the function $z = f(x, y)$ which is 0 if $x < y$ and 1 otherwise. They would begin by secret sharing x and y with each other over an appropriate ring (i.e., P_0 creates $\langle x\rangle^\ell = (\langle x\rangle^\ell_0, \langle x\rangle^\ell_1)$ with $x = \langle x\rangle^\ell_0 + \langle x\rangle^\ell_1 \bmod 2^\ell$ and gives P_1 the share $\langle x\rangle^\ell_1$ while P_1 similarly creates shares $(\langle y\rangle^\ell_0, \langle y\rangle^\ell_1)$ and gives P_0 the share $\langle y\rangle^\ell_0$). Next, they would execute an interactive protocol Π_f that would compute random shares of z, i.e., at the end of this protocol P_0 and P_1 would hold (say boolean) shares $\langle z\rangle^B_0$ and $\langle z\rangle^B_1$ such that $\langle z\rangle^B_0 \oplus \langle z\rangle^B_1 = z$. Π_f comes with a guarantee that P_0 learns nothing about y and P_1 learns nothing about x by this interaction; if the parties wish to learn z, then they can do so by reconstructing the 2 shares (e.g. P_0 can send the share $\langle z\rangle^B_0$ to P_1 who computes z).

Security Model. Our threat model is same as [27] and considers 2-party computation secure against a static PPT semi-honest adversary. At the beginning of the protocol, the adversary corrupts one of the two parties, and tries to learn information about honest party's input using the protocol messages while faithfully following the protocol. We argue security using the standard simulation paradigm [19], [34], [52].

Our protocols for floating-point invoke several 2PC sub-protocols that realize the building blocks described below and satisfy the same security definition (see Section VII).

D. 2PC Building Blocks

Table I describes the building blocks we use and their communication cost (refer Appendix G for details), and includes the mixed-bitwidth 2PC functions that are realizable using the protocols from [63], [73]. For instance, the protocol for multiplexer for ℓ-bit values is denoted by Π^ℓ_{MUX}. It takes as input boolean shares of conditional c and ℓ-bit arithmetic shares of two values x and y and returns ℓ-bit arithmetic shares of z, where $z = x$ if $c = 1$ and $z = y$ otherwise. We use the protocol from [64] with total communication $2\lambda + 2\ell$. Below, we define two new building blocks used by our functionalities and show how to realize them in Appendix G2.

a) Less-than-and-equal: $\mathsf{LT\&EQ}(x, y)$ takes $x, y \in \mathbb{Z}_{2^\ell}$ as input, and outputs two bits $c_{\mathsf{LT}} = \mathbf{1}\{x < y\}$ and $c_{\mathsf{EQ}} = \mathbf{1}\{x = y\}$. The advantage of using this functionality (over comparison and equality separately) is that one can implement it at nearly the same cost as one comparison. We provide a protocol $\Pi^\ell_{\mathsf{LT\&EQ}}$ that takes as input $\langle x\rangle^\ell, \langle y\rangle^\ell$ and securely computes $\langle c_{\mathsf{LT}}\rangle^B, \langle c_{\mathsf{EQ}}\rangle^B$ with $\lambda(\ell + 3) + 14\ell + 60$ bits of communication.

b) Truncate-and-Reduce-with-Sticky-bit: $\mathsf{TRS}(x, s)$ is similar to truncate-and-reduce (Table I), except that it additionally sets the least-significant-bit of the output as 1 if any of the lower s bits of x is 1. We provide a protocol $\Pi^{\ell,s}_{\mathsf{TRS}}$ that takes as input $\langle x\rangle^\ell$ and securely computes $\langle \mathsf{TRS}(x, s)\rangle^{\ell-s}$ with $\lambda(s + 5) + \ell + 14s + 60$ bits of communication.

V. PRIMITIVE OPERATIONS

In the following, we use Greek letters for floating-point numbers and Roman letters for fixed-point numbers or integers. We denote a floating-point number α, parameterized with $p, q \in \mathbb{Z}^+$, as a tuple (z, s, e, m) where $z \in \{0, 1\}$ is the zero-bit (set if $\alpha = 0$), $s \in \{0, 1\}$ is the sign-bit (set only if $\alpha \le 0$), $e \in \{0, 1\}^{p+2}$ is the (unbiased) signed exponent taking values in the range $[-2^{p-1}+1, 2^{p-1}]$ and $m \in \{0, 1\}^{q+1}$ is the (normalized) unsigned fixed-point mantissa taking values from $[2^q, 2^{q+1} - 1] \cup \{0\}$ with scale q. We note that $\alpha = (z, s, e, m)$ represents the real number $(1-z)\cdot(1-2s)\cdot 2^{\mathsf{int}_{p+2}(e)} \cdot [\![m]\!]_{q+1,q}$. Below, we write $\alpha.z$ for the z component of the α tuple, and similarly for other components. For single-precision floating-point numbers, recall that the IEEE standard sets $p = 8$ and $q = 23$. Following IEEE, we define the infinities special values $(-1)^s\mathtt{inf}$ as $(0, s, 2^{p-1}, 2^q)$. Similarly, the special values $(-1)^s 0$ are defined as $(1, s, -2^{p-1} + 1, 0)$. During operations, when the result overflows, i.e., the magnitude of the result becomes greater than the largest floating-point number, the result is mapped to $\pm\mathtt{inf}$. Similarly, when the result underflows, that is, its magnitude becomes too small to be represented as floating-point numbers, then the result is mapped to ± 0. We provide the functionality $\mathcal{F}^{p,q}_{\mathsf{FPCheck}}$ that checks for overflows and underflows in Figure 1. Note that the

Functionality $\mathcal{F}^{p,q}_{\mathsf{FPCheck}}(\alpha)$

1: $\alpha = (z, s, e, m)$
2: **if** $\mathbf{1}\{e > 2^{p-1} - 1\}$ **then**
3: $\quad m = 2^q$; $e = 2^{p-1}$
4: **if** $\mathbf{1}\{z = 1\} \vee \mathbf{1}\{e < 2 - 2^{p-1}\}$ **then**
5: $\quad m = 0$; $e = 1 - 2^{p-1}$; $z = 1$
6: Return (z, s, e, m)

Fig. 1: *Checking for overflows and underflows.*

Functionality $\mathcal{F}^{\ell}_{\mathsf{RNTE}}(x, r)$

1: $a = \mathsf{TRS}(x, r-2)$; $\mathsf{idx} = a \bmod 8$
2: $c = L_{\mathsf{RNTE}}(\mathsf{idx})$, $c \in \{0, 1\}$
3: Return $\mathsf{TR}(a, 2) + \mathsf{ZXt}(c, \ell - r)$

Fig. 2: *Round Nearest Ties to Even:* $x \gg_R r$.

underflow check also ensures that the mantissa and exponent are consistent with the zero-bit.

Correctness. For all the primitive operators, we provide the following correctness guarantee. For inputs that are normalized floating-point numbers, the outputs of our functionality are bitwise equivalent to the primitive operators in the IEEE standard, except if the outputs are denormals or NANs. Our baselines that we compare against provide the same or weaker correctness guarantees (Section III). The pen-and-paper correctness proofs are available in the full version of this paper [62]. In the following, we provide the functionalities in detail and the high-level intuition behind them.

We begin by describing the functionality for rounding that is used by all our primitive operators and math functions. Steps in rounding and primitive operations functionalities can be securely realized using building blocks from Section IV-D (see Section VII). In this section, we describe floating-point multiplication, addition and division and defer the details of comparison to Appendix H. We show how to mechanically translate these functionalities to 2PC protocols in Section VII.

A. Rounding $\gg_R$

The rounding-nearest (ties to even) operator $x \gg_R r$ takes as input an ℓ-bit fixed-point number x with scale s, and returns a $(\ell - r)$-bit fixed-point number y with scale $(s - r)$ that is closest to x in terms of the real value (Section III-B). In concrete terms, we have $|[\![y]\!]_{\ell-r,s-r} - [\![x]\!]_{\ell,s}| \leq 2^{-(s-r)-1}$. We provide the functionality for rounding in Figure 2.

Correctness: Let $x = x_{\ell-r-1}||d||g||x_{r-1}$, where $x_{\ell-r-1} \in \{0,1\}^{\ell-r-1}$, $d, g \in \{0,1\}$, $x_{r-1} \in \{0,1\}^{r-1}$. First, we replace x_{r-1} with a sticky bit f which is set if any bit of x_{r-1} is set using TRS. Now, for correct results we need to round up when g and f are set (round up result is closer than the round down result). For the tie case, i.e., when f is zero and g is set, if d is set (number is odd) then we round it up to even. We use a lookup table (LUT) $L_{\mathsf{RNTE}}(d||g||f) = g \wedge (d \vee f)$ to encode these cases and realize round up by $\mathsf{TR}(x, r) + 1$.

Functionality $\mathcal{F}^{p,q,Q}_{\mathsf{Round}^*}(e, m)$

1: **if** $\mathbf{1}\{m \geq 2^{Q+1} - 2^{Q-q-1}\}$ **then**
2: $\quad$ Return $(e + 1, 2^q)$
3: **else**
4: $\quad$ Return $(e, m \gg_R (Q - q))$

Fig. 3: *Round mantissa and check for overflow.*

B. Round&Check $\mathcal{F}^{p,q,Q}_{\mathsf{Round}^*}$

In floating-point operations, rounding is usually performed on a normalized mantissa $m \in [2^Q, 2^{Q+1})$ of $Q + 1$ bits to reduce its fractional bits from higher precision Q to the lower precision of the floating-point representation q. Thus, the mantissa is rounded by $Q - q$ bits and we get an output mantissa of $q+1$ bits. During rounding, a normalized mantissa, however, can become unnormalized ($= 2^{q+1}$) and overflow $q + 1$ bits, in case the input mantissa is close to 2^{Q+1} and is rounded up. To account for this, in case of overflows, the mantissa is set to 2^q and the exponent e is incremented to adjust for dividing mantissa by 2. We denote this functionality by $\mathcal{F}^{p,q,Q}_{\mathsf{Round}^*}(e, m)$ and describe it in Figure 3.

Correctness: First note that m is only rounded to 2^{q+1} if $m \geq 2^{Q+1} - 2^{Q-q-1}$. In case $m > 2^{Q+1} - 2^{Q-q-1}$, $\frac{m}{2^{Q-q}}$ is closer to 2^{q+1} than $2^{q+1} - 1$. If $m < 2^{Q+1} - 2^{Q-q-1}$, $\frac{m}{2^{Q-q}}$ is closer to $2^{q+1} - 1$. If $m = 2^{Q+1} - 2^{Q-q-1}$, according to the defintion of round-ties-to-even, we choose the even representation, i.e., 2^{q+1}. Now, in case there are no overflows, we simply round the mantissa and return the result Step 4. Otherwise, we increment the exponent by 1 and divide the rounded mantissa by 2 to get 2^{q+1} Step 2.

C. Multiplication $\boxdot_{p,q}$

Given two floating-point values α_1 and α_2, we can write their (exact) product α as $(1 - \alpha.z) \cdot (1 - 2\alpha.s) \cdot 2^{\mathsf{int}(\alpha_1.e)+\mathsf{int}(\alpha_2.e)} \cdot [\![\alpha_1.m]\!]_{q+1,q} \cdot [\![\alpha_2.m]\!]_{q+1,q}$, where $\alpha.z = \alpha_1.z \vee \alpha_2.z$ and $\alpha.s = \alpha_1.s \oplus \alpha_2.s$. Thus, we first add the input exponents to get $e = \alpha_1.e + \alpha_2.e$. Next, we multiply the input mantissas, which results in a $2q + 2$-bit fixed-point integer m with scale $2q$ such that $[\![m]\!]_{2q+2,2q} \in [1, 4)$. Now, we normalize the mantissa and adjust the exponent to create a normalized floating-point output. There are two cases: Roughly, when $[\![m]\!]_{2q+2,2q} < 2$, we simply round it by q bits and set e as the output exponent. Otherwise, we round m by $q+1$ bits and set the output exponent as $e+1$. Finally, we check for overflow and underflow. We describe our multiplication functionality $\mathcal{F}_{\mathsf{FPMul}}$ formally in Figure 4. Additionally, we perform the optimization of Appendix I1.

D. Addition $\boxplus_{p,q}$

We need to add two floating-point numbers, α_1 and α_2. Let β_1 be the larger of the two numbers in magnitude and β_2 be the smaller one. Next, we compute the difference of their exponents, $d = \beta_1.e - \beta_2.e$. Now we have the following 2 cases: (1) $d > q + 1$: we simply return β_1 as β_2 is too small compared to β_1. (2) $d \leq q + 1$: we align the exponents of

Functionality $\mathcal{F}^{p,q}_{\mathsf{FPMul}}(\alpha_1, \alpha_2)$

1: $e = \alpha_1.e + \alpha_2.e$
2: $m = \alpha_1.m *_{2q+2} \alpha_2.m$
3: **if** $\mathbf{1}\{m < 2^{2q+1} - 2^{q-1}\}$ **then**
4: $\quad m = m \gg_R q \bmod 2^{q+1}$
5: **else**
6: $\quad m = m \gg_R (q+1);\ e = e + 1$
7: $s = \alpha_1.s \oplus \alpha_2.s;\ z = \alpha_1.z \vee \alpha_2.z$
8: Return $\alpha = \mathcal{F}^{p,q}_{\mathsf{FPCheck}}(z, s, e, m)$

Fig. 4: *Floating-Point Multiplication:* $\alpha_1 \boxdot_{p,q} \alpha_2$

Functionality $\mathcal{F}^{p,q}_{\mathsf{FPAdd}}(\alpha_1, \alpha_2)$

1: $(e_{\mathsf{LT}}, e_{\mathsf{EQ}}) = \mathsf{LT\&EQ}(\alpha_1.e, \alpha_2.e)$
2: $m_{\mathsf{LT}} = \mathbf{1}\{\alpha_1.m < \alpha_2.m\}$
3: $(\beta_1, \beta_2) = e_{\mathsf{LT}} \oplus (e_{\mathsf{EQ}} \wedge m_{\mathsf{LT}})\ ?\ (\alpha_2, \alpha_1) : (\alpha_1, \alpha_2)$
4: $d = \beta_1.e - \beta_2.e$
5: **if** $\mathbf{1}\{d > q+1\}$ **then**
6: $\quad$ Return β_1
7: **else**
8: $\quad m_1 = \beta_1.m *_{2q+2} 2^d$
9: $\quad m_2 = \mathsf{ZXt}(\beta_2.m, 2q+2)$
10: $\quad m_2 = (\beta_1.s \oplus \beta_2.s\ ?\ -m_2 : m_2)$
11: $\quad m = m_1 + m_2;\ e = \beta_2.e$
12: $\quad k, K = \mathsf{MSNZB}(m), K = 2^{2q+1-k}$
13: $\quad m = m *_{2q+2} K;\ e = e + k - q$
14: $\quad (e, m) = \mathcal{F}^{p,q,2q+1}_{\mathsf{Round}^*}(e, m).$
15: $\quad z = \mathbf{1}\{m = 0\};\ s = \beta_1.s$
16: $\quad$ Return $\alpha = \mathcal{F}^{p,q}_{\mathsf{FPCheck}}(z, s, e, m)$

Fig. 5: *Floating-Point Addition:* $\alpha_1 \boxplus_{p,q} \alpha_2$

both operands to $\beta_2.e$ by left-shifting $\beta_1.m$ by d to obtain the mantissa m_1 with bitlength $2q+2$ and scale q. Next, we add/subtract $\beta_2.m$ from m_1 depending on sign of β_2. With this, we have an unnormalized mantissa m with bitlength $2q+3$ and scale q such that $\beta_2.e$ is the corresponding exponent. To normalize m, we left-shift it by $2q+1-k$ bits, where k is most significant non-zero bit (MSNZB) of m, and set its scale as $2q+1$. Accordingly, we subtract $2q+1-k$ from the exponent to account for the left-shift and add $q+1$ to it for the scale change. To reduce the scale of mantissa from $2q+1$ to q bits, we round it using $\mathcal{F}^{p,q,2q+1}_{\mathsf{Round}^*}$ from Section V-B to get the mantissa in $q+1$ bits. The sign of the output is the same as the sign of β_1 and we check if m is 0 to set the zero-bit. Our floating-point addition functionality $\mathcal{F}_{\mathsf{FPAdd}}$ is given in Figure 5.

E. Division $\boxslash_{p,q}$

Given numerator α_1 and denominator α_2, we can write their (exact) division α as $\alpha_1.z \cdot (1-2\alpha.s) \cdot 2^{\mathsf{int}_{p+2}(\alpha_1.e) - \mathsf{int}_{p+2}(\alpha_2.e)} \cdot (\llbracket \alpha_1.m \rrbracket_{q+1,q} / \llbracket \alpha_2.m \rrbracket_{q+1,q})$, where $\alpha.s = \alpha_1.s \oplus \alpha_2.s$. Thus, we first subtract the exponents to get $e = \alpha_1.e - \alpha_2.e$. Next, we compare the mantissas and if $\alpha_1.m < \alpha_2.m$, we double $\alpha_1.m$ to get m_1 and subtract 1 from the output exponent e. This ensures that the quotient of the mantissas is always normalized. To calculate the quotient, we first compute an approximation r_0 for the reciprocal of $\alpha_2.m$ with relative error $< 2^{-g}$ by using a lookup table of size 2^g (Step 6). Then, we

Functionality $\mathcal{F}^{p,q}_{\mathsf{FPDiv}}(\alpha_1, \alpha_2)$

1: $m_1 = \mathsf{ZXt}(\alpha_1.m, q+2);\ m_2 = \alpha_2.m;\ e = \alpha_1.e - \alpha_2.e$
2: **if** $\mathbf{1}\{\alpha_1.m < \alpha_2.m\}$ **then**
3: $\quad m_1 = 2m_1;\ e = e - 1$
4: $t = 2;\ g = \lceil \frac{q+1}{2^t} \rceil + 1;\ k_0 = g + 1$
5: $h = \mathsf{TR}(m_2, q-g) \bmod 2^g$
6: $r_0 = L_{\mathsf{recp-init}}(h), r_0 \in \{0,1\}^{k_0+2}$
7: **for** $i = 1$ to t **do**
8: $\quad k_i = 2^i \cdot (g-1) + 3$
9: $\quad f_i = 2^{k_i} - \mathsf{TR}(m_2 *_{k_{i-1}+q+1} r_{i-1}, q + k_{i-1} - k_i)$
10: $\quad r_i = r_{i-1} *_{k_i+2} 2^{k_i - k_{i-1}} + \mathsf{TR}(r_{i-1} *'_{k_i+k_{i-1}+2} f_i, k_{i-1})$
11: $m'' = m_1 *_{k_t+q+2} r_t;\ m' = \mathsf{TR}(m'', k_t)$
12: $y_1 = m_2 *_{q+3} m';\ y_2 = y_1 + \mathsf{ZXt}(m_2, q+3)$
13: $y = (m_1 *_{q+3} 2^{q+1}) - (y_1 + y_2);\ (\mathsf{lt}, \mathsf{eq}) = \mathsf{LT\&EQ}(0, y)$
14: $m = \mathsf{lt} \oplus (\mathsf{eq} \wedge \mathbf{1}\{m \bmod 2 = 1\})\ ?\ m' + 1 : m'$
15: $s = \alpha_1.s \oplus \alpha_2.s,\ z = \alpha_1.z$
16: Return $\alpha = \mathcal{F}^{p,q}_{\mathsf{FPCheck}}(z, s, e, m)$

Fig. 6: *Floating-Point Division:* $\alpha_1 \boxslash_{p,q} \alpha_2$.

perform t Newton iterations on this approximation (Steps 7–10), essentially doubling the precision with each iteration, to get r_t with relative error $< 2^{-(2^t(g-1)+1)} \leq 2^{-q-2}$. Next, we multiply r_t with m_1 to get an approximation of the quotient m' that has an ULP error of at most 1 (Step 11). At this point, we show that the correct quotient can either be $\llbracket m' \rrbracket_{q+1,q}$ or $\llbracket m'+1 \rrbracket_{q+1,q}$. We decide between the two (in Steps 12–14) by checking whether $\llbracket m_1 \rrbracket_{q+2,q}$ is closer to $\llbracket m' \cdot \alpha_2.m \rrbracket_{2q+3,2q}$ or $\llbracket (m'+1) \cdot \alpha_2.m \rrbracket_{2q+3,2q}$. Finally, the output sign-bit is set as the XOR of the input sign-bits, the output zero-bit is set as the numerator's zero-bit, and we perform checks for underflows and overflows.

VI. Math functions

We describe our implementations for math functions over floats, i.e., $p = 8$ and $q = 23$. For SecFloat's design of math functions please see Section I-D and Section II. Recall that the math functions in standard math libraries follow a generic three step process: *range reduction*, *polynomial approximations*, and *output compensation*.

Next, we describe the operators that the math functionalities use. We use floating-point addition and multiplication with various parametrizations of (p, q) in polynomial approximation and output compensation to obtain accurate results. We observe that for some of the intermediate operations, we don't need correctly rounded results to meet the 1 ULP error bound. Here, the intermediate operands have a restricted range and the checks for corner cases can be omitted. In particular, we omit the calls to $\mathcal{F}_{\mathsf{FPCheck}}$. Also, we have created functionalities for cheap addition (denoted by $\boxplus^*$) and cheap division (denoted by $\boxslash^*$) that do *not* return correctly rounded results and describe them in Appendix I. Hence, the functionalities use $\boxplus/\boxdot$ (without $\mathcal{F}_{\mathsf{FPCheck}}$) and cheap addition/division while ensuring that the final result is precise. We avoid calls to floating-point comparisons of the form $\alpha < \beta$ by designing the functionalities so that β is usually a power of 2 and the comparison can be done by just comparing exponents.

Correctness. To prove the precision of our functionalities, we exhaustively test on all possible floating-point inputs. For each input α, we compute β the floating-point output of the functionality. We also soundly model the exact real result using high-precision arithmetic, i.e., MPFR [32] with 500-bits of precision. MPFR produces a tiny range $[r_1, r_2]$ in which the exact real result must lie. We compute a sound bound on the ULP error of β using $[r_1, r_2]$, i.e, $\max(\text{ULP}(\beta, r_1), \text{ULP}(\beta, r_2))$. This bound is below 1 for all our math functionalities on all inputs. Evaluating a functionality on a single input takes microseconds and we can test each functionality on all inputs within a few hours.

In the following, we use $\texttt{Float}_{p,q}(r)$ to denote the floating-point number, parametrized by p and q, obtained by correctly rounding the real number r.

A. Spline Evaluation

All our math functionalities use *splines*, or piecewise polynomials for polynomial approximation. An n-piece spline F is parameterized by the *knots* $\mathcal{K} = \{\kappa_1, \ldots, \kappa_{n+1}\}$ and n polynomials of degree d. Let the set of coefficients of these polynomials be $\Theta = \{\theta_i^{(j)}\}_{i=0,j=1}^{d,n}$. Then, spline F computed on $\delta \in [\kappa_1, \kappa_{n+1})$ is

$$F(\delta) = \theta_0^{(j)} + \theta_1^{(j)} \cdot \delta + \ldots + \theta_d^{(j)} \cdot \delta^d, \text{ for } \delta \in [\kappa_j, \kappa_{j+1})$$

The main task is to select the correct polynomial coefficients for δ from Θ. In all our math functions, we set up the knots $\mathcal{K}$ such that the *active interval* for δ (j s.t. $\delta \in [\kappa_j, \kappa_{j+1})$) can be determined using only a few bits from the exponent and the mantissa of δ. We use these bits, ℓ in number, to define $\mathsf{idx} \in \{0,1\}^\ell$ from δ and $K = \{k_1, \ldots, k_{n+1}\}$ from $\mathcal{K}$ with $k_j \in \{0,1\}^\ell$ for all $j \in [n+1]$ such that $\delta \in [\kappa_j, \kappa_{j+1})$ if and only if $\mathsf{idx} \in [k_j, k_{j+1}]$. With this the task is to select correct coefficients given idx, Θ, K. In Appendix J, we provide a new specialized crypto-friendly functionality for this which has up to $3.9\times$ lower communication than the baselines for the splines occuring in SECFLOAT's functionalities.

Next, we describe the math functionalities for trigonometric sine and logarithm. Exponentiation is in Appendix A; tangent is in Appendix B; cosine is in Appendix C. The polynomials are evaluated using Horner's rule, e.g., $a+bx+cx^2$ is evaluated as $a + x(b + cx)$.

B. Sine

We provide the functionality for computing $\mathsf{sin}\pi(\alpha)$ in Figure 7. First, we handle the easy cases in Steps 3-7.

- If the input $|\alpha| \geq 2^{23}$, we know that $\mathsf{sin}\pi(\alpha) = 0$ as all such numbers are integers.
- If the input $|\alpha| < 2^{-14}$, it is small enough that approximating $\mathsf{sin}\pi(\alpha)$ as $\pi\alpha$ leads to less than 1 ULP error.

For other inputs, we perform the range reduction:

1) Range Reduction : We use the property that $\mathsf{sin}\pi$ is an odd periodic function to reduce the computation of $\mathsf{sin}\pi(\alpha)$ where $|\alpha| \in [2^{-14}, 2^{23})$ to $\mathsf{sin}\pi(\delta)$ where $\delta \in [0, 0.5]$. Let $s = \mathbf{1}\{\alpha < 0\}$ and $\beta = |\alpha| = 2 \cdot K + a + n$, where $K \in \mathbb{N}$, $a \in \{0,1\}$ and $n \in [0,1)$. Now, consider $\delta \in [0, 0.5]$, defined as $\delta = n$ if $n < 0.5$ and $\delta = 1 - n$ otherwise. Then, using $\mathsf{sin}\pi(1-x) = \mathsf{sin}\pi(x)$, we have:

$$\mathsf{sin}\pi(\alpha) = (-1)^{a+s} \cdot \mathsf{sin}\pi(\delta) \tag{1}$$

Although this equation is over reals, we compute bits $a, s \in \{0,1\}$ and a reduced input $\delta \in [0, 0.5]$ from α using the building blocks of Section IV-D in Steps 9-14.

2) Polynomial Evaluation (Steps 15-28): For reduced inputs $\delta < 2^{-14}$, we simply approximate $\mathsf{sin}\pi(\delta)$ as $\pi \cdot \delta$. The computation of $\mathsf{sin}\pi(\delta)$ on the rest of the inputs is split into two cases, and we design a separate spline of degree 5 (of the form $\theta_1\delta + \theta_3\delta^3 + \theta_5\delta^5$) for both cases. The first case considers $\delta \in [2^{-14}, \frac{1}{32})$ and we use a 9 piece spline F^1 here, whose intervals are determined by the lower 4 bits of the exponent. The second case handles $\delta \in [\frac{1}{32}, 0.5]$ and here we use a 34 piece spline F^2. Lower 2 bits of exponent and upper 5 bits of mantissa suffice to determine the intervals of F^2 on all inputs except $\delta = 0.5$. To avoid looking at additional bits for $\delta = 0.5$, in Step 24, if $\delta = 0.5$ we set the 7-bit index idx_2 to the idx_2 corresponding to the last interval $[\frac{63}{128}, 0.5)$.

3) Output Compensation (Step 29): By Equation 1, we return $\mathsf{sin}\pi(\delta)$ if $a \oplus s$ is 0, and $-\mathsf{sin}\pi(\delta)$ otherwise.

Functionality $\mathcal{F}^{8,23}_{\mathsf{FPsin}\pi}(\alpha)$

1: $p = 8$; $q = 23$; $Q = 27$
2: $\alpha' = (\alpha.z, 0, \alpha.e, \alpha.m *_{Q+1} 2^{Q-q})$
3: **if** $\mathbf{1}\{\alpha.e \geq 23\}$ **then**
4: Return $\texttt{Float}_{p,q}(0)$
5: **else if** $\mathbf{1}\{\alpha.e < -14\}$ **then**
6: $\gamma = \texttt{Float}_{p,Q}(\pi) \boxdot_{p,Q} \alpha'$
7: Return $(\gamma.z, \gamma.s, \mathcal{F}^{p,q,Q}_{\mathsf{Round}*}(\gamma.e, \gamma.m))$
8: **else**
9: $m = \alpha.m *_{q+15} 2^{\alpha.e+14}$
10: $a = \mathsf{TR}(m, q+14)$; $n = m \bmod 2^{q+14}$
11: $f = (\mathbf{1}\{n > 2^{q+13}\} \ ? \ 2^{q+14} - n : n)$
12: $k, K = \mathsf{MSNZB}(f)$; $f = f *_{q+15} K$
13: $z = \mathbf{1}\{f = 0\}$; $e = (z \ ? \ -2^{p-1} + 1 : k - q - 14)$
14: $\delta = (z, 0, e, \mathsf{TR}(f, q + 14 - Q))$
15: **if** $\mathbf{1}\{\delta.e < -14\}$ **then**
16: $\mu = \texttt{Float}_{p,Q}(\pi) \boxdot_{p,Q} \delta$
17: **else**
18: **if** $\mathbf{1}\{\delta.e < -5\}$ **then**
19: $\mathsf{idx}_1 = \delta.e + 14 \bmod 2^4$
20: $(\theta_1, \theta_3, \theta_5) = \mathsf{GetC}_{4,9,5,p,Q}(\mathsf{idx}_1, \Theta^1_{\mathsf{sin}}, K^1_{\mathsf{sin}})$
21: **else**
22: $\mathsf{idx}_2 = 32 \cdot (\delta.e + 5 \bmod 2^7)$
23: $\mathsf{idx}_2 = \mathsf{idx}_2 + \mathsf{ZXt}(\mathsf{TR}(\delta.m, Q-5) \bmod 32, 7)$
24: $\mathsf{idx}_2 = \mathbf{1}\{\delta.e = -1\} \ ? \ 127 : \mathsf{idx}_2$
25: $(\theta_1, \theta_3, \theta_5) = \mathsf{GetC}_{7,34,5,p,Q}(\mathsf{idx}_2, \Theta^2_{\mathsf{sin}}, K^2_{\mathsf{sin}})$
26: $\Delta = \delta \boxdot_{p,Q} \delta$
27: $\mu = ((\theta_5 \boxdot_{p,Q} \Delta) \boxplus^*_{p,Q} \theta_3) \boxdot_{p,Q} \Delta$
28: $\mu = (\mu \boxplus^*_{p,Q} \theta_1) \boxdot_{p,Q} \delta$
29: Return $(\mu.z, a \oplus \alpha.s, \mathcal{F}^{p,q,Q}_{\mathsf{Round}*}(\mu.e, \mu.m))$.

Fig. 7: *Floating-point* $\mathsf{sin}\pi_{8,23}(\alpha)$.

C. Logarithm

The functionality for computing $\log_2(\alpha)$ is given in Figure 8. The input $\alpha \in [2^{-126}, 2^{128})$, i.e., $\alpha.s = \alpha.z = 0$ and

Functionality $\mathcal{F}^{8,23}_{\mathsf{FPlog2}}(\alpha)$

$p = 8$; $q = 23$; $Q = 27$
$a = \mathbf{1}\{\alpha.e = -1\}$; $f = a\ ?\ 2^{q+1} - \alpha.m : \alpha.m - 2^q$
$k, K = \mathsf{MSNZB}(f)$; $f = f *_{q+1} K$
$e = a\ ?\ k - q - 1 : k - q$
$z = \mathbf{1}\{f = 0\}$; $e = (z\ ?\ -2^{p-1} + 1 : e)$
$N = \alpha.e$; $\delta = (z, 0, e, f *_{Q+1} 2^{Q-q})$
if $\mathbf{1}\{\delta.z\}$ **then**
  $\mu = \mathtt{Float}_{p,Q}(0)$
else
  if $\mathbf{1}\{\delta.e < -5\}$ **then**
    $\mathsf{idx}_1 = \delta.e + 24 \bmod 2^5$
    $(\theta_0^a, \theta_1^a, \theta_2^a, \theta_3^a) = \mathsf{GetC}_{5,19,4,p,Q}(\mathsf{idx}_1, \Theta^1_{\mathsf{log}}, K^1_{\mathsf{log}})$
    $(\theta_0^b, \theta_1^b, \theta_2^b, \theta_3^b) = \mathsf{GetC}_{5,18,4,p,Q}(\mathsf{idx}_1, \Theta^3_{\mathsf{log}}, K^3_{\mathsf{log}})$
  else
    $\mathsf{idx}_2 = 16 \cdot (\delta.e + 5 \bmod 2^7)$
    $\mathsf{idx}_2 = \mathsf{idx}_2 + \mathsf{ZXt}(\mathsf{TR}(\delta.m, Q-4) \bmod 16, 7)$
    $(\theta_0^a, \theta_1^a, \theta_2^a, \theta_3^a) = \mathsf{GetC}_{7,20,4,p,Q}(\mathsf{idx}_2, \Theta^2_{\mathsf{log}}, K^2_{\mathsf{log}})$
    $(\theta_0^b, \theta_1^b, \theta_2^b, \theta_3^b) = \mathsf{GetC}_{7,32,4,p,Q}(\mathsf{idx}_2, \Theta^4_{\mathsf{log}}, K^4_{\mathsf{log}})$
  $(\theta_0, \theta_1, \theta_2, \theta_3) = a\ ?\ (\theta_0^a, \theta_1^a, \theta_2^a, \theta_3^a) : (\theta_0^b, \theta_1^b, \theta_2^b, \theta_3^b)$
  $\mu = ((\theta_3 \boxdot_{p,Q} \delta) \boxplus^*_{p,Q} \theta_2) \boxdot_{p,Q} \delta$
  $\mu = ((\mu \boxplus^*_{p,Q} \theta_1) \boxdot_{p,Q} \delta) \boxplus^*_{p,Q} \theta_0$
$\beta = L_{\mathsf{Int2Float}}(N), N \in \{0,1\}^p$
$\beta' = (\beta.z, \beta.s, \beta.e, \beta.m *_{Q+1} 2^{Q-6})$
$\gamma = a\ ?\ \mu : \mu \boxplus^*_{p,Q} \beta'$
Return $(\gamma.z, \gamma.s, \mathcal{F}^{p,q,Q}_{\mathsf{Round}*}(\gamma.e, \gamma.m))$

Fig. 8: *Floating-point* $\mathsf{log2}_{8,23}(\alpha)$.

we proceed with following steps.

1) Range Reduction (Steps 2-6): Let $\alpha = m \cdot 2^N$, where $m \in [1, 2)$ and $N \in [-126, 127] \cap \mathbb{Z}$. Let $a = \mathbf{1}\{N = -1\}$ and define δ as $\delta = m - 1$ if $a = 0$ and $\delta = 1 - \frac{m}{2}$ if $a = 1$. Then, it is easy to see that the following holds:

$$\log_2(\alpha) = \begin{cases} N + \log_2(1+\delta) & a = 0 \\ \log_2(1-\delta) & a = 1 \end{cases} \quad (2)$$

Thus, from the range reduction, we get a bit $a \in \{0,1\}$, a reduced input $\delta \in [0,1)$, and an integer $N \in \{0,1\}^8$. We use this more complicated range reduction than $\log_2(\alpha) = N + \log_2(1+\delta)$ as the latter suffers from cancellation errors when $N = -1$ and δ is close to 1. Hence, the requirement of precise results and desire to keep the bitwidths small dictate that we use Equation 2.

2) Polynomial Evaluation (Steps 7-21): Depending on the value of a, we need to compute either $\log_2(1+\delta)$ or $\log_2(1-\delta)$, where $\delta \in [0,1)$. If $\delta = 0$, we simply return 0. For other inputs, we found that $\delta \in [2^{-23}, 1)$ when $a = 0$ and $\delta \in [2^{-24}, 0.5]$ when $a = 1$. For all the cases described below, we use degree-3 splines.

When $a = 1$, we split the computation of $\log_2(1-\delta)$ in two cases and design separate splines for both. 1) $\delta \in [2^{-24}, \frac{1}{32})$: a 19-piece spline F^1 such that intervals are determined by the lower 5 bits of the exponent. 2) $\delta \in [\frac{1}{32}, 0.5]$: a 20-piece spline F^2 where intervals are determined using 7 bits: lower 3 bits of exponent and upper 4 bits of mantissa.

When $a = 0$, to compute $\log_2(1+\delta)$, we do similar two cases. 1) $\delta \in [2^{-23}, \frac{1}{32})$: a 18 piece spline F^3 determined by the same lower 5 bits of exponent. 2) $\delta \in [\frac{1}{32}, 1)$: a 32-piece spline F^4 determined by the same 7 bits as above.

We provide details on exact knots used in Appendix L. As the knots used in the two cases of $a = 0$ and $a = 1$ have a large overlap, in GetC, we combine the first look-up step in F^1 and F^3 (similarly, F^2 and F^4) to reduce its cost.

3) Output Compensation (Steps 23-25): Let μ be the output of the polynomial evaluation. As per Equation 2, if $a = 1$, we simply return μ; else, we convert N to a floating-point number β with $p = 8$ and $q = 6$ using an LUT $L_{\mathsf{Int2Float}}$ of size 256, convert β to floating-point with $p = 8$ and $q = 27$, and round $\mu + \beta$ to $q = 23$.

Protocol $\Pi^{p,q}_{\mathsf{FPMul}}(\langle\alpha_1\rangle^{\mathsf{FP}(p,q)}, \langle\alpha_2\rangle^{\mathsf{FP}(p,q)})$

Set $\langle e\rangle^{p+2} = \langle\alpha_1.e\rangle^{p+2} + \langle\alpha_2.e\rangle^{p+2}$.
Call $\langle m'\rangle^{2q+2} = \Pi^{q+1,q+1,2q+2}_{\mathsf{UMult}}(\langle\alpha_1.m\rangle^{q+1}, \langle\alpha_2.m\rangle^{q+1})$.
Call $\langle c\rangle^B = \Pi^{2q+2}_{\mathsf{GT}}(2^{2q+1} - 2^{q-1}, \langle m\rangle^{2q+2})$.
Call $\langle m_1\rangle^{q+1} = \Pi^{2q+2,q}_{\mathsf{RNTE}}(\langle m'\rangle^{2q+2}) \bmod 2^{q+1}$.
Call $\langle m_2\rangle^{q+1} = \Pi^{2q+2,q+1}_{\mathsf{RNTE}}(\langle m'\rangle^{2q+2})$.
Set $\langle e_2\rangle^{p+2} = \langle e\rangle^{p+2} + 1$.
Call $\langle m\rangle^{q+1} = \Pi^{q+1}_{\mathsf{MUX}}(\langle c\rangle^B, \langle m_1\rangle^{q+1}, \langle m_2\rangle^{q+1})$.
Call $\langle e\rangle^{p+2} = \Pi^{p+2}_{\mathsf{MUX}}(\langle c\rangle^B, \langle e\rangle^{p+2}, \langle e_2\rangle^{p+2})$.
Set $\langle s\rangle^B = \langle\alpha_1.s\rangle^B \oplus \langle\alpha_2.s\rangle^B$.
Call $\langle z\rangle^B = \Pi_{\mathsf{OR}}(\langle\alpha_1.z\rangle^B, \langle\alpha_2.z\rangle^B)$.
Ret. $\langle\beta\rangle^{\mathsf{FP}(p,q)} = \Pi^{p,q}_{\mathsf{FPCheck}}(\langle z\rangle^B, \langle s\rangle^B, \langle e\rangle^{p+2}, \langle m\rangle^{q+1})$.

Fig. 9: *2PC protocol for floating-point multiplication*

VII. 2PC PROTOCOLS FOR FLOATING POINT

For ease of exposition, in Sections V and VI we described the crypto-friendly cleartext functionalities for floating-point operations used in SECFLOAT. Here, we discuss how we can obtain the corresponding 2PC protocols using the building blocks discussed in Section IV-D in a straight-forward manner. First, consistent with the floating-point representation used by our functionalities, a secret shared floating-point number α is a tuple $\langle\alpha\rangle^{\mathsf{FP}(p,q)} = (\langle\alpha.z\rangle^B, \langle\alpha.s\rangle^B, \langle\alpha.e\rangle^{p+2}, \langle\alpha.m\rangle^{q+1})$ of four secret-shared values.

Next, as an example, we provide the 2PC protocol for floating-point multiplication in Figure 9 derived from the cleartext functionality in Figure 4. Here, parties start with secret-shares of 2 floating-point values α_1 and α_2 and end up with shares of $\alpha_1 \boxdot_{p,q} \alpha_2$. For each step in Figure 4, we compute the shares of the output from the shares of the input by invoking the correct 2PC building block corresponding to the computation in that step. For instance, shares of the value m', i.e., the product of the mantissas $\alpha_1.m$ and $\alpha_2.m$, are obtained using a call to the protocol for unsigned multiplication over shares. All the conditionals or "if" statements are executed using a call to the multiplexer protocol that works over boolean shares of the conditional variable and arithmetic shares of the values being chosen. This transforms the data dependent flow in the cleartext functionality to data oblivious computation in the 2PC protocol. For example, Steps 7 and 8 correspond to the conditional setting of m and e in Steps 3-6 in Figure 4. Apart

Protocol $\Pi^{p,q}_{\mathsf{FPCheck}}(\langle\alpha\rangle^{\mathsf{FP}(p,q)})$

1: Call $\langle c_1\rangle^B = \Pi^{p+2}_{\mathsf{GT}}(\langle\alpha.e\rangle^{p+2}, 2^{p-1}-1)$.
2: Call $\langle m\rangle^{q+1} = \Pi^{q+1}_{\mathsf{MUX}}(\langle c_1\rangle^B, 2^q, \langle\alpha.m\rangle^{q+1})$.
3: Call $\langle e\rangle^{p+2} = \Pi^{p+2}_{\mathsf{MUX}}(\langle c_1\rangle^B, 2^{p-1}, \langle\alpha.e\rangle^{p+2})$.
4: Call $\langle c_2\rangle^B = \Pi^{p+2}_{\mathsf{GT}}(2-2^{p-1}, \langle\alpha.e\rangle^{p+2})$.
5: Call $\langle c_3\rangle^B = \Pi_{\mathsf{OR}}(\langle\alpha.z\rangle^B, \langle c_2\rangle^B)$.
6: Call $\langle m\rangle^{q+1} = \Pi^{q+1}_{\mathsf{MUX}}(\langle c_3\rangle^B, 0, \langle m\rangle^{q+1})$.
7: Call $\langle e\rangle^{p+2} = \Pi^{p+2}_{\mathsf{MUX}}(\langle c_3\rangle^B, 1-2^{p-1}, \langle e\rangle^{p+2})$.
8: Call $\langle z\rangle^B = \Pi^B_{\mathsf{MUX}}(\langle c_3\rangle^B, 1, \langle\alpha.z\rangle^B)$.
9: Return $\langle\beta\rangle^{\mathsf{FP}(p,q)} = (\langle z\rangle^B, \langle\alpha.s\rangle^B, \langle e\rangle^{p+2}, \langle m\rangle^{q+1})$.

Fig. 10: *2PC protocol for checking overflows and underflows*

Protocol $\Pi^{\ell,r}_{\mathsf{RNTE}}(\langle x\rangle^\ell)$

1: Call $\langle a\rangle^{\ell-r+2} = \Pi^{\ell,r-2}_{\mathsf{TRS}}(\langle x\rangle^\ell)$.
2: Set $\langle \mathsf{idx}\rangle^3 = \langle a\rangle^{\ell-r+2} \bmod 8$.
3: Call $\langle c\rangle^B = \Pi^{3,1}_{\mathsf{LUT}}(L_{\mathsf{RNTE}}, \langle \mathsf{idx}\rangle^3)$.
4: Call $\langle a\rangle^{\ell-r} = \Pi^{\ell-r+2,2}_{\mathsf{TR}}(\langle a\rangle^{\ell-r+2})$.
5: Call $\langle c\rangle^{\ell-r} = \Pi^{1,\ell-r}_{\mathsf{ZXt}}(\langle c\rangle^B)$.
6: Return $\langle a\rangle^{\ell-r} + \langle c\rangle^{\ell-r}$.

Fig. 11: *2PC protocol for Round Nearest Ties to Even*

from the calls to the 2PC building blocks in Section IV-D, the 2PC protocol for floating-point multiplication also invokes protocols that check for underflows and overflows (Π_{FPCheck}) and round nearest ties to even (Π_{RNTE}). For completeness, we provide these protocols in Figures 10 and 11, respectively, that are derived using similar transformations to cleartext functionalities in Figures 1 and 2.

We can follow the same mechanical procedure to obtain the 2PC protocols for all primitive floating point operations from the corresponding cleartext functionalities in Section V. Moreover, for the math functionalities in Section VI, each of the steps can be securely computed using either the protocol for the floating-point primitive operations (or, its cheaper variant as discussed), or the building blocks from Section IV-D.

As is standard, security of the overall protocols for floating-point operations can be argued in the *hybrid model* [19] as follows. First, we replace the calls to sub-protocols with calls to corresponding ideal (trusted) functionalities. Second, we note that all the sub-protocols/ideal functionalities maintain the invariant that they take 2-out-of-2 secret shares of the inputs and produce 2-out-of-2 secret shares of the output. Hence, during simulation, it is secure to provide random values (from the correct domain) as outputs of these ideal functionalities. With these observations, simulation is straight-forward. For instance, we prove security of our protocol for floating-point multiplication against an adversary that corrupts P_0 as follows: the simulator will provide uniform random values from appropriate domains as shares for m', c, m_1, m_2, m, e, z and $\beta.z, \beta.s, \beta.e, \beta, m$. It holds that the views of the adversary in the real execution and this simulated execution are identical. Proofs of security for all our protocols are similar.

VIII. SecFloat implementation

We have designed SecFloat as a drop-in replacement for floating-point libraries. The users of SecFloat can take their existing numerical software written in C++ and link it with SecFloat instead of standard math libraries to obtain secure 2PC implementations. SecFloat is implemented on top of SIRNN's [63] building blocks available at [5] using 3200 lines of C++. Additionally, [63] provided an optimization, *MSB-to-Wrap*, that significantly reduces the cost of building blocks when the most significant bit (MSB) of the input is known (See Table 5, [63]). To use this optimization, in our implementation of floating-point functionalities, many of the bitwidths are incremented by 1 to keep track of the MSB. Appendix K describes how to obtain splines for the math functions.

IX. Evaluation

We empirically evaluate SecFloat and show that it significantly advances the state-of-the-art in 2PC of floating-point computation both in terms of precision, as well as communication and latency. We compare against three baselines:

First, we compare with ABY [1] and EMP [2], two state-of-the-art cryptographic libraries with floating-point support. They use heavily optimized circuits that were generated automatically from hardware synthesis tools in ABY-F [27]. Here, on some inputs, all 23 bits of mantissa are garbage and even some bits of the exponent are incorrect. In contrast, SecFloat implementations are both provably accurate (for primitive operations and math functions) and have up to $30\times$ lower communication (Section IX-A2).

For our second baseline, we compare with the MP-SPDZ [9], [42] semi-honest 2PC implementations of the custom functionalities for primitive floating-point operations presented by Aliasgari *et al.* [10], [11]. Unlike SecFloat, these implementations are imprecise (six orders of magnitude higher errors) and suffer up to $240\times$ higher communication overheads (Section IX-A2). MP-SPDZ doesn't support floating-point math functions.

As a result, the only way to evaluate math functions like sine precisely using 2PC frameworks that exist today is to take the C-code of *precise* math implementations from an open source math library like OpenLibm [48] and run it using 2PC with a compiler that can translate C-code to 2PC protocols. This is our third baseline. Like SecFloat, this baseline is guaranteed to be *precise* (OpenLibm is *precise* and 2PC protocols provide bitwise equivalent results to cleartext). However the communication overheads for this baseline are intractable. For example, for sine, SecFloat's communication is at least $1000\times$ lower than this baseline (Section IX-A3). Finally, performance of state-of-the-art SilentOT [18], [26], [72] based protocols is $4.5 - 1152\times$ worse than SecFloat (Appendix F).

Finally, we evaluate the SecFloat library with two applications: privacy-preserving proximity testing (Appendix M) and inference in privacy-preserving advertisements (Section IX-B).

Experimental Setup. We perform our experiments on two machines, each equipped with a 4-core 3.7 GHz Xeon pro-

cessor and 16 GBs of RAM. The machines are connected via LAN network providing 3 Gbps bandwidth and 0.8 ms RTT.

A. Microbenchmarks and Comparison

In Table II, we consider the benchmarks[4] from [27], i.e., floating-point comparison (CMP), multiplication (MULT), addition (ADD), division (DIV), $\sin\pi$ (SIN), $\cos\pi$ (COS), $\tan\pi$ (TAN), 2^{α} (EXP2), and $\log_2 \alpha$ (LOG2). Next, we evaluate SECFLOAT and the baselines.

1) SECFLOAT *evaluation:* As can be seen from the last column, SECFLOAT has ideal ULP error: half for primitive operations and one for math functions. The penultimate column shows the communication in kilobytes per operation. We evaluate SECFLOAT with different batch sizes of 1000 to 100,000 operations and report the total execution time. For batches below size 1000, the time of SECFLOAT is about the same as the time for batch of size 1000. This unexpected behavior is because of engineering inefficiencies in the source code of SIRNN [5], [63] that SECFLOAT currently uses for its building blocks. In particular, if the implementations in SIRNN were to be parallelized then the cost of small batches would be lower. Since improvements in the 2PC implementations of the building blocks is orthogonal to SECFLOAT, for this evaluation we use the SIRNN implementations as is.

2) Imprecise baselines: We run the microbenchmarks[5] with ABY-F circuits [27]. For each ABY-F row in Table II, the numbers shown are from evaluating these circuits from [27] with EMP. ABY's garbled circuits and ABY's GMW perform worse and we compare against them in Appendix E. We observe that the math functions (except for EXP2) have huge ULP errors. For example, on running LOG2 with input just below one, SECFLOAT and MKL output $-8.59913 \cdot 10^{-8}$ and ABY-F outputs $-1.19209 \cdot 10^{-7}$, which is incorrect in every digit and has a relative error of 40%. We show such examples for other math functions in Appendix D. SECFLOAT's communication is also up to $30\times$ lower than ABY-F, resulting in $46\times$ reduction in running time. Table III shows that the rounds of SECFLOAT are comparable to the rounds of ABY-F with the GMW backend. We note that we report analytical rounds for SECFLOAT here, and empirically, our code has up to $4\times$ more rounds due to implementation artifacts. For machine learning Section IX-B), the 2PC performance is bottlenecked by the communication and not the rounds.

We evaluate the functions available in MP-SPDZ (based on [10], [11]) in Table II. These implementations are imprecise and SECFLOAT outperforms them in both precision (up to $10^6\times$) and communication (up to $240\times$).

3) Precise baselines: Since the trigonometric functions are imprecise in ABY-F circuits [27], and are unavailable in MP-SPDZ, the only option today to run them with 2PC precisely is to compile existing *precise* C-implementations to 2PC. We use OpenLibm as it has open-source C-code[6] for *precise* math functions. The only C-to-2PC compiler that can handle OpenLibm's implementation for trigonometric sine is CBMC-GC, the state-of-the-art compiler from C to garbled circuits [13], [39], [61]. We evaluate this CBMC-GC+OpenLibm approach for trigonometric sine. Here, CBMC-GC generates a Boolean circuit with 1.7 million Non-XOR gates and running such a circuit using any available implementation of Yao's garbled circuits will incur a communication of at least 51MiB for each evaluation [1], [2], [9]. This is three orders of magnitude larger than the communication of our $\sin\pi$ functionality and quickly becomes intractable.

Next, we consider Berkeley's SoftFloat [37] as a baseline, which is the gold standard for correct implementations of

Technique	Time (in s) for Batch Size			Comm.	ULP
	10^3	10^4	10^5	(KiB)	Error
CMP					
SECFLOAT	0.02	0.07	0.30	1.11	NA
ABY-F	0.05 (2.3x)	0.25 (3.9x)	2.08 (7.0x)	6.93 (6.2x)	NA
MP SPDZ	0.22 (10.7x)	1.24 (19.0x)	12.85 (43.1x)	89.64 (80.7x)	NA
MULT					
SECFLOAT	0.05	0.14	0.69	3.13	0.5
ABY-F	0.34 (7.0x)	3.22 (23.0x)	32.20 (46.7x)	95.74 (30.6x)	0.5
MP SPDZ	0.38 (7.7x)	1.36 (9.7x)	10.28 (14.9x)	72.72 (23.2x)	$> 10^6$
ADD					
SECFLOAT	0.15	0.56	3.91	11.10	0.5
ABY-F	0.12 (0.8x)	1.00 (1.8x)	9.35 (2.4x)	31.33 (2.8x)	0.5
MP SPDZ	3.63 (23.8x)	36.05 (64.4x)	– –	2561.22 (230.7x)	$> 10^6$
DIV					
SECFLOAT	0.14	0.35	2.43	10.27	0.5
ABY-F	0.46 (3.2x)	3.90 (11.2x)	38.36 (15.8x)	119.55 (11.6x)	0.5
MP SPDZ	11.33 (78.4x)	113.25 (326.3x)	– –	2462.68 (239.8x)	$> 10^6$
SIN					
SECFLOAT	0.46	1.59	11.27	42.36	1
ABY-F	0.48 (1.0x)	4.49 (2.8x)	44.77 (4.0x)	163.49 (3.9x)	$> 10^7$
COS					
SECFLOAT	0.50	1.60	11.29	42.41	1
ABY-F	0.46 (0.9x)	4.51 (2.8x)	45.21 (4.0x)	163.86 (3.9x)	$> 10^7$
TAN					
SECFLOAT	0.66	2.21	14.91	60.38	1
ABY-F	1.32 (2.0x)	12.39 (5.6x)	122.37 (8.2x)	445.86 (7.4x)	$> 10^7$
EXP2					
SECFLOAT	0.41	1.51	11.28	37.23	1
ABY-F	0.85 (2.0x)	8.38 (5.6x)	83.61 (7.4x)	304.89 (8.2x)	1
LOG2					
SECFLOAT	0.53	1.97	14.94	51.48	1
ABY-F	0.92 (1.7x)	9.04 (4.6x)	90.53 (6.1x)	330.77 (6.4x)	$> 10^6$

Table II: Comparison of SECFLOAT with ABY-F and MP-SPDZ. The numbers in parentheses show our improvement factor. '-' denotes tool crash.

[4]SECFLOAT also supports *correct* square root and *precise* e^x, $\log_e x$, etc.

[5]TAN(x) has been implemented as DIV(SIN(x), COS(x)) for ABY-F.

[6]Intel MKL's implementations are in handwritten x86-assembly and there are no compilers from x86-assembly to 2PC.

OP	Rounds		OP	Rounds	
	ABY-F	SECFLOAT		ABY-F	SECFLOAT
CMP	12	11	SIN/COS	95/98	196
MULT	47	27	TAN	394	268
ADD	59	69	EXP2	100	187
DIV	296	84	LOG2	157	256

Table III: Rounds comparison of SECFLOAT with ABY-F.

Task	Size	Time (s)	Comm. (GB)	Improvement
Backprop.	1	40.3	10.9	11.3×
Backprop.	32	1020.5	329.6	9.1×
Inference	1	18.3	4.7	8.8×
Inference	32	1036.0	151.58	8.8×

Table IV: Secure machine learning with SECFLOAT and its communication improvements over ABY-F.

primitive operations[7]. For this comparison, we implemented SoftFloat's functionalities with the same building blocks as SECFLOAT, and found that SoftFloat's functionalities have $6.1\times, 2.4\times, 3.2\times$, and $2.4\times$ worse communication than SECFLOAT for comparison, multiplication, addition, and division, respectively. SECFLOAT's improvement can be largely attributed to the use of non-standard bitwidths (SoftFloat uses only the standard bitwidths of $\{8, 16, 32, 64\}$) and fewer "if-then-else" statements. For instance, SoftFloat's addition has different branches depending on the equality of signs of operand, and then further two more sub-branches depending on which operand is larger. In contrast, SECFLOAT runs a uniform computation for all four cases.

B. Machine Learning

Consider the problem of web advertising, that is witnessing a push for 2PC-based solutions, through proposals like Google's DOVEKEY [4] and Microsoft's MACAW [8]. Commercial advertisement networks use deep neural networks (DNNs) for the *relevance* problem; given the user information known as context and an advertisement, the model outputs a score indicating whether the advertisement is relevant to the user or not. There are two parties—Alice holding a secret unlabeled input and Bob holding the secret model weights—who want to run secure inference to learn relevance scores without revealing their secrets to each other.

The prior works on secure deep learning [17], [23], [31], [45], [47], [53], [56], [58], [60] (and references therein) rely on converting floating-point models to fixed-point models. They require the scale or precision level of fixed-point numbers to be set manually (often ≤ 16). Low scales cause precision loss and high scales cause integer overflows. Hence, a good scale that *preserves* model accuracy (i.e., where the accuracy of fixed-point model is close to that of the floating-point model) might or might not exist. To determine the best scale, CRYPTFLOW [47], [63], [64] enumerates fixed-point models with all possible scales and picks the most accurate model.

In this section, we evaluate an industrial relevance model $\mathcal{M}$. It is a fully-connected feed-forward neural network that takes a vector of length 874 as input. It has three hidden layers with sizes 300, 200, and 100, respectively, and each hidden layer is followed by a ReLU activation. The output layer has 4 classes with softmax activation. We describe the protocol for ReLU in Appendix N, and the rest of the operations are easy to compute by composing primitive and math operations. We evaluated $\mathcal{M}$ with CRYPTFLOW and found that no good scales exists for $\mathcal{M}$ because $\mathcal{M}$ has weights in a large dynamic range: they are as large as 10 and as small as 10^{-7} in magnitude. The dynamic range of intermediate activations is even larger and this large dynamic range of values makes fixed-point approximations unsuitable for $\mathcal{M}$. All the fixed-point programs, each with a different scale, generated by CRYPTFLOW for $\mathcal{M}$, either overflow the integers for large values or underflow small values to zero. In both cases, the fixed-point model output is garbage.

In Table IV, we perform secure inference of $\mathcal{M}$ with SECFLOAT and report the runtime and communication for minibatch sizes 1 and 32. For completeness, we also evaluate backpropagation: one forward pass, one backward pass while using the cross entropy loss, and a weight update. This model uses additions, multiplications, comparisons, divisions, and exponentiations. As expected, the communication of SECFLOAT-based implementations are much lower than those based on ABY-F (last column of Table IV). Hence, SECFLOAT enables accurate secure inference of relevance models in privacy-preserving advertising, which is impossible with the existing fixed-point-based frameworks.

X. CONCLUSION AND FUTURE WORK

We build a library SECFLOAT for 2PC of floats which is up to six orders of magnitude more precise and up to three orders of magnitude more efficient than ABY-F, CBMC-GC, and MP-SPDZ. Going forward, we would like to provide security against malicious adversaries and math functions for doubles. The former is an engineering effort that requires running SECFLOAT's crypto-friendly functionalities with SPDZ [42]. Accurate math functions over doubles require advanced verification techniques (e.g., [35], [36], [50]) as verifying the precision on all inputs via exhaustive testing on $\approx 2^{64}$ doubles is intractable. With SECFLOAT, we make a case to end the use of ad hoc float-to-fixed conversions in secure inference. In the future, we would like to explore what benefits SECFLOAT brings to other applications in privacy-preserving machine learning.

ACKNOWLEDGMENT

We thank Pratik Bhatu for ML frontend support and Yuval Ishai for helpful feedback.

REFERENCES

[1] "ABY: A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," https://github.com/encryptogroup/ABY, 2015.

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

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

[7] SoftFloat doesn't have math functions.

[4] " Dovekey auction using Secure Multi Party Computation (MPC)," https://github.com/google/ads-privacy/blob/master/proposals/dovekey/dovekey_auction.md, 2021.

[5] "CrypTFlow: An End-to-end System for Secure TensorFlow Inference," https://github.com/mpc-msri/EzPC, 2021.

[6] "Intel®C++ Compiler Classic Developer Guide and Reference ," https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/intel-math-library/overview-intel-math-library.html, 2021.

[7] "Intel®oneAPI Math Kernel Library Vector Mathematics Performance and Accuracy Data ," https://software.intel.com/content/www/us/en/develop/documentation/onemkl-vmperfdata/top.html, 2021.

[8] "Multi-party Computation of Ads on the Web (MaCAW)," https://github.com/WICG/privacy-preserving-ads/blob/main/MACAW.md, 2021.

[9] "Multi-Protocol SPDZ: Versatile framework for multi-party computation (accessed: May 2021)," 2021. [Online]. Available: https://github.com/data61/MP-SPDZ

[10] M. Aliasgari, M. Blanton, and F. Bayatbabolghani, "Secure computation of hidden markov models and secure floating-point arithmetic in the malicious model," *Int. J. Inf. Sec.*, 2017.

[11] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele, "Secure computation on floating point numbers," in *NDSS*, 2013.

[12] A. Aly and N. P. Smart, "Benchmarking privacy preserving scientific operations," in *ACNS*, 2019.

[13] D. W. Archer, S. Atapoor, and N. P. Smart, "The Cost of IEEE Arithmetic in Secure Computation," Cryptology ePrint Archive, Report 2021/054, 2021.

[14] S. Arita and S. Nakasato, "Fully homomorphic encryption for point numbers," in *Inscrypt*, ser. Lecture Notes in Computer Science, 2016.

[15] S. Bai, G. Yang, J. Shi, G. Liu, and Z. Min, "Privacy-preserving oriented floating-point number fully homomorphic encryption scheme," *Secur. Commun. Networks*, 2018.

[16] G. R. Blakley, "Safeguarding cryptographic keys," in *International Workshop on Managing Requirements Knowledge*, 1979.

[17] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "MP2ML: a mixed-protocol machine learning framework for private inference," in *ARES*, 2020.

[18] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round OT extension and silent non-interactive secure computation," in *CCS*, 2019.

[19] R. Canetti, "Security and Composition of Multiparty Cryptographic Protocols," *J. Cryptology*, 2000.

[20] O. Catrina, "Round-efficient protocols for secure multiparty fixed-point arithmetic," in *COMM*, 2018.

[21] ——, "Evaluation of floating-point arithmetic protocols based on shamir secret sharing," in *ICETE (Selected Papers)*, 2019.

[22] ——, "Performance analysis of secure floating-point sums and dot products," in *COMM*, 2020.

[23] N. Chandran, D. Gupta, S. L. B. Obbattu, and A. Shah, "SIMC: ML inference secure against malicious clients at Semi-Honest cost," in *USENIX Security*, 2022.

[24] V. Chen, V. Pastro, and M. Raykova, "Secure Computation for Machine Learning With SPDZ," in *Workshop on Privacy Preserving Machine Learning at NeurIPS*, 2018.

[25] W. J. Cody and W. Waite, *Software Manual for the Elementary Functions (Prentice-Hall Series in Computational Mathematics)*. USA: Prentice-Hall, Inc., 1980.

[26] G. Couteau, S. Raghuraman, and P. Rindal, "Silver: Silent VOLE and Oblivious Transfer from Hardness of Decoding Structured LDPC Codes," in *CRYPTO 2021*.

[27] D. Demmler, G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, and S. Zeitouni, "Automated synthesis of optimized circuits for secure computation," in *CCS*, 2015.

[28] D. Demmler, T. Schneider, and M. Zohner, "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," in *NDSS*, 2015.

[29] G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner, "Pushing the Communication Barrier in Secure Computation using Lookup Tables," in *NDSS*, 2017.

[30] V. Dimitrov, L. Kerik, T. Krips, J. Randmets, and J. Willemson, "Alternative implementations of secure real numbers," in *CCS*, 2016.

[31] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits," in *CRYPTO*, 2020.

[32] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, 2007.

[33] D. Goldberg, "What every computer scientist should know about floating-point arithmetic." *ACM Comput. Surv.*, vol. 23, no. 1, 1991.

[34] O. Goldreich, S. Micali, and A. Wigderson, "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority," in *ACM STOC*, 1987.

[35] J. Harrison, "Floating point verification in HOL light: The exponential function," *Formal Methods Syst. Des.*, vol. 16, no. 3, 2000.

[36] ——, "Formal verification of floating point trigonometric functions," in *FMCAD*, ser. Lecture Notes in Computer Science, vol. 1954, 2000.

[37] J. R. Hauser, "Berkeley softfloat (release 3e)," http://www.jhauser.us/arithmetic/SoftFloat.html, 2018.

[38] B. Hemenway, S. Lu, R. Ostrovsky, and W. W. IV, "High-precision secure computation of satellite collision probabilities," in *SCN*, 2016.

[39] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *CCS*, 2012.

[40] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending Oblivious Transfers Efficiently," in *CRYPTO 2003*.

[41] L. Kamm and J. Willemson, "Secure floating point arithmetic and private satellite collision analysis," *Int. J. Inf. Sec.*, vol. 14, no. 6, 2015.

[42] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *CCS*, 2020.

[43] L. Kerik, P. Laud, and J. Randmets, "Optimizing MPC for robust and scalable integer and floating-point arithmetic," in *Financial Cryptography Workshops*, 2016.

[44] D. Kim, Y. Son, D. Kim, A. Kim, S. Hong, and J. H. Cheon, "Privacy-preserving approximate gwas computation based on homomorphic encryption," *BMC Medical Genomics*, vol. 13, 2020.

[45] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure multi-party computation meets machine learning," in *NeurIPS*, 2021.

[46] T. Krips and J. Willemson, "Hybrid model of fixed and floating point numbers in secure multiparty computations," in *ISC*, 2014.

[47] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *IEEE Symposium on Security and Privacy*, 2020.

[48] J. Lang, "OpenLibm," https://openlibm.org/, 2016.

[49] W. Lee, R. Sharma, and A. Aiken, "Verifying bit-manipulations of floating-point," in *PLDI*, 2016.

[50] ——, "On automatically proving the correctness of math.h implementations," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 2018.

[51] J. P. Lim and S. Nagarakatte, "RLIBM-32: high performance correctly rounded math libraries for 32-bit floating point representations," in *PLDI*, 2021.

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

[53] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious Neural Network Predictions via MiniONN Transformations," in *CCS*, 2017.

[54] X. Liu, R. H. Deng, W. Ding, R. Lu, and B. Qin, "Privacy-preserving outsourced calculation on floating point numbers," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 11, 2016.

[55] Y. Liu, Y. Chiang, T. Hsu, C. Liau, and D. Wang, "Floating point arithmetic protocols for constructing secure data analysis application," in *KES*, ser. Procedia Computer Science, 2013.

[56] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A Cryptographic Inference Service for Neural Networks," in *USENIX Security*, 2020.

[57] P. Mohassel and P. Rindal, "ABY^3: A Mixed Protocol Framework for Machine Learning," in *CCS*, 2018.

[58] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," in *IEEE S&P*, 2017.

[59] S. Moon and Y. Lee, "An efficient encrypted floating-point representation using HEAAN and TFHE," *Secur. Commun. Networks*, 2020.

[60] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: improved mixed-protocol secure two-party computation," in *USENIX Security*, 2021.

[61] P. Pullonen and S. Siim, "Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations," in *Financial Cryptography Workshops*, 2015.

[62] D. Rathee, A. Bhattacharya, R. Sharma, D. Gupta, N. Chandran, and A. Rastogi, "Secfloat: Accurate floating-point meets secure 2-party computation (full version)," Cryptology ePrint Archive, Report 2022/322, 2022, https://ia.cr/2022/322.
[63] D. Rathee, M. Rathee, R. K. K. Goli, D. Gupta, R. Sharma, N. Chandran, and A. Rastogi, "SIRNN: A math library for secure inference of RNNs," in *IEEE S&P*, 2021.
[64] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow2: Practical 2-Party Secure Inference," in *CCS*, 2020.
[65] J. F. Reiser and D. E. Knuth, "Evading the drift in floating-point addition," *Inf. Process. Lett.*, vol. 3, no. 3, 1975.
[66] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "DeepSecure: Scalable Provably-Secure Deep Learning," in *DAC*, 2018.
[67] A. Shamir, "How to share a secret," *Commun. ACM*, 1979.
[68] R. W. Sinnott, "Virtues of the Haversine," *Sky Telesc.*, vol. 68, 1984.
[69] P. H. Sterbenz, *Floating-point computation*, ser. Prentice-Hall series in automatic computation. Prentice-Hall, 1974.
[70] J. Šeděnka and P. Gasti, "Privacy-preserving distance computation and proximity testing on earth, done right," in *AsiaCCS*, 2014.
[71] Xilinx, "Vivado Design Suite User Guide, High-level Synthesis," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf, 2015.
[72] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, "Ferret: Fast extension for correlated OT with small communication," in *CCS*, 2020.
[73] A. C. Yao, "How to Generate and Exchange Secrets (Extended Abstract)," in *FOCS*, 1986.
[74] T. Zhu, X. Zou, and J. Pan, "Query with SUM aggregate function on encrypted floating-point numbers in cloud," *J. Inf. Process. Syst.*, vol. 13, no. 3, 2017.

APPENDIX

A. Exponentiation

Figure 12 describes the functionality to compute 2^α. First, we handle the easy cases in Steps 2-7.

- If $\alpha \geq 128$, we return `inf` as 2^α overflows 32-bit floats.
- If $\alpha < -126$, we return 0 as 2^α underflows.
- If $|\alpha| < 2^{-24}$, then returning 1 results in < 1 ULP error.

1) Range Reduction (Steps 9-13): Let $s = \mathbf{1}\{\alpha < 0\}$ and $\beta = |\alpha| = K + \delta$, $K \in \mathbb{N}$ and $\delta \in [0, 1)$. For $N = (-1)^s \cdot K$, $2^\alpha = 2^N \cdot 2^{-\delta}$ if $s = 1$ and $2^\alpha = 2^N \cdot 2^\delta$ otherwise. From this range reduction, we get a bit $s \in \{0, 1\}$, an integer $N \in \mathbb{Z} \cap [-126, 128)$, and a reduced input $\delta \in [0, 1)$.

2) Polynomial Evaluation (Steps 14-23): Based on the sign-bit s, we need to compute either 2^δ or $2^{-\delta}$ where $\delta \in [0, 1)$. If $\delta < 2^{-24}$ then we can again return 1. For the remaining inputs, we use two 64-piece degree-2 splines, F^+ for 2^δ and F^- for $2^{-\delta}$. Both F^+ and F^- have identical and equidistant knots that are multiples of $\frac{1}{64}$. For small δ with $\delta.e \in [-24, -7]$, it is okay to set $\delta.e = -7$ without affecting the active interval of δ. With this, to determine the active interval of δ, we only need to examine 8 bits of δ, namely, bottom 3 bits of exponent and top 5 bits of mantissa. We retrieve the corresponding coefficients for both F^+ and F^- for the active interval, assign the coefficients of the quadratic polynomial $\theta_0 + \theta_1\delta + \theta_2\delta^2$ based on s, and evaluate this polynomial on δ.

3) Output Compensation (Step 24): The final output is simply $2^N \cdot \mu$, where μ is the output of polynomial evaluation. Note that since N is an integer, this multiplication is equivalent to adding N to μ's exponent.

Functionality $\mathcal{F}^{8,23}_{\mathsf{FPexp2}}(\alpha)$

1: $p = 8$; $q = 23$; $Q = 27$; $\beta = (\alpha.z, 0, \alpha.e, \alpha.m)$
2: **if** $\mathbf{1}\{\alpha.s = 0\} \wedge \mathbf{1}\{\alpha.e \geq 7\}$ **then**
3: Return `inf`
4: **else if** $\mathbf{1}\{\alpha <_{p,q} \mathtt{Float}_{p,q}(-126)\}$ **then**
5: Return $\mathtt{Float}_{p,q}(0)$
6: **else if** $\mathbf{1}\{\alpha.e < -24\}$ **then**
7: Return $\mathtt{Float}_{p,q}(1)$
8: **else**
9: $m = \alpha.m *_{q+32} 2^{\alpha.e+24}$
10: $s = \alpha.s$; $f = m \bmod 2^{q+24}$; $N = \mathsf{TR}(m, q + 24)$
11: $k, K = \mathsf{MSNZB}(f)$; $f = \mathsf{TR}(f *_{q+25} K, 24 + q - Q)$
12: $z = \mathbf{1}\{f = 0\}$; $e = (z\ ?\ -2^{p-1} + 1 : k - q - 24)$
13: $N = (s\ ?\ -N : N)$; $\delta = (z, 0, e, f)$
14: **if** $\mathbf{1}\{\delta.e < -24\}$ **then**
15: $\mu = \mathtt{Float}_{p,Q}(1)$
16: **else**
17: $e' = \mathbf{1}\{\delta.e < -6\}\ ?\ -7 : \delta.e$
18: $\mathsf{idx} = 32 \cdot (e' + 7 \bmod 2^8)$; $\Delta = \delta \boxdot_{p,Q} \delta$
19: $\mathsf{idx} = \mathsf{idx} + \mathsf{ZXt}(\mathsf{TR}(\delta.m, Q - 5) \bmod 32, 8)$
20: $(\theta_0^+, \theta_1^+, \theta_2^+) = \mathsf{GetC}_{8,64,3,p,Q}(\mathsf{idx}, \Theta^+_{\mathsf{exp2}}, K_{\mathsf{exp2}})$
21: $(\theta_0^-, \theta_1^-, \theta_2^-) = \mathsf{GetC}_{8,64,3,p,Q}(\mathsf{idx}, \Theta^-_{\mathsf{exp2}}, K_{\mathsf{exp2}})$
22: $(\theta_0, \theta_1, \theta_2) = s\ ?\ (\theta_0^-, \theta_1^-, \theta_2^-) : (\theta_0^+, \theta_1^+, \theta_2^+)$
23: $\mu = (((\theta_2 \boxdot_{p,Q} \delta) \boxplus^*_{p,Q} \theta_1) \boxdot_{p,Q} \delta) \boxplus^*_{p,Q} \theta_0$
24: Return $(0, 0, \mathcal{F}^{p,q,Q}_{\mathsf{Round}^*}(\mathsf{ZXt}(N, p + 2) + \mu.e, \mu.m))$

Fig. 12: *Floating-point* $\mathsf{exp2}_{8,23}(\alpha)$.

B. Tangent

We provide the functionality for computing $\mathsf{tan}\pi(\alpha)$ in Figure 13. For $|\alpha| \notin [2^{-14}, 2^{23})$, $\mathsf{tan}\pi(\alpha) = \mathsf{sin}\pi(\alpha)$ (Steps 3-7). For $\alpha \in [2^{-14}, 2^{23})$, we perform the following steps.

1) Range Reduction: Standard approaches for range reduction of $\mathsf{tan}\pi$ use high degree polynomials that is not crypto-friendly. So, we designed our own range reduction for $\mathsf{tan}\pi$ as follows. Similar to $\mathsf{sin}\pi$, $\mathsf{tan}\pi$ is also an odd periodic function. Let $s = \mathbf{1}\{\alpha < 0\}$ and $\beta = |\alpha| = K + m$, $K \in \mathbb{N}$ and $m \in [0, 1)$. Now, $\mathsf{tan}\pi(m) = (-1)^b \cdot \mathsf{tan}\pi(n)$, where $b = \mathbf{1}\{m > 0.5\}$ and $n = b\ ?\ 1 - m : m$. We have $\mathsf{tan}\pi(n) = \mathsf{tan}\pi(t)^{1-2c}$, where $c = \mathbf{1}\{m > 0.25\}$ and $t = c\ ?\ 0.5 - n : n$. Now, we rewrite $t = \frac{N}{512} + \delta$, where $N \in \{0, \ldots, 127\}$ and $\delta \in [0, \frac{1}{512}]$. Finally, using the identity $\mathsf{tan}\pi(x + y) = \frac{\mathsf{tan}\pi(x) + \mathsf{tan}\pi(y)}{1 - \mathsf{tan}\pi(x) \cdot \mathsf{tan}\pi(y)}$, we get:

$$\mathsf{tan}\pi(\alpha) = (-1)^{b+s} \cdot \Big(\frac{\mathsf{tan}\pi(\tfrac{N}{512}) + \mathsf{tan}\pi(\delta)}{1 - \mathsf{tan}\pi(\tfrac{N}{512}) \cdot \mathsf{tan}\pi(\delta)}\Big)^{1-2c} \quad (3)$$

Thus, we get bits $b, c, s \in \{0, 1\}$, an index $N \in \{0, 1\}^7$ and a reduced input $\delta \in [0, \frac{1}{512}]$ in Steps 9-21.

2) Polynomial Evaluation (Steps 22-28): We use a 21-piece spline with polynomials of the form $\theta_1\delta + \theta_3\delta^3$ to compute $\mathsf{tan}\pi(\delta)$ for $\delta \in [2^{-14}, \frac{1}{512}]$. We use 3 bits of the exponent and 2 bits from the mantissa of δ to lookup the coefficients from the LUT $L_{\mathsf{tan\text{-}coeffs}}$ of size 21.

3) Output Compensation (Steps 29-33): $\mathsf{tan}\pi(\frac{N}{512})$ is retrieved from a lookup table of size 128 and the output is computed according to Equation 3.

Functionality $\mathcal{F}^{8,23}_{\mathsf{FPtan}\pi}(\alpha)$

$p = 8$; $q = 23$; $Q = 27$
$\alpha' = (\alpha.z, 0, \alpha.e, \alpha.m *_{Q+1} 2^{Q-q})$
if $\mathbf{1}\{\alpha.e \geq 23\}$ **then**
  Return $\mathtt{Float}_{p,q}(0)$
else if $\mathbf{1}\{\alpha.e < -14\}$ **then**
  $\gamma = \mathtt{Float}_{p,Q}(\pi) \boxdot_{p,Q} \alpha'$
  Return $(\gamma.z, \gamma.s, \mathcal{F}^{p,q,Q}_{\mathsf{Round}^*}(\gamma.e, \gamma.m))$
else
  if $\mathbf{1}\{\alpha.e \geq -9\}$ **then**
    $m = \alpha.m *_{q+9} 2^{\alpha.e+9}$
    $(n, b) = \mathbf{1}\{m > 2^{q+8}\}$? $(2^{q+9} - m, 1) : (m, 0)$
    $(t, c) = \mathbf{1}\{n > 2^{q+7}\}$? $(2^{q+8} - n, 1) : (n, 0)$
    $f = t \bmod 2^q$; $N = \mathsf{TR}(t, q)$
    if $\mathbf{1}\{N = 2^7\}$ **then**
      $N = N - 1 \bmod 2^7$; $\delta' = \mathtt{Float}_{p,q}(2^{-9})$
    else
      $k, K = \mathsf{MSNZB}(f)$; $f = f *_{q+1} K$
      $z = \mathbf{1}\{f = 0\}$; $e = (z$? $-2^{p-1}+1 : k-q-9)$
      $N = N \bmod 2^7$; $\delta' = (z, 0, e, f)$
  else
    $b = 0$; $c = 0$; $N = 0$; $\delta' = (\alpha.z, 0, \alpha.e, \alpha.m)$
  $\delta = (\delta'.z, \delta'.s, \delta'.e, \delta'.m *_{Q+1} 2^{Q-q})$
  if $\mathbf{1}\{\delta.e < -14\}$ **then**
    $\mu = \mathtt{Float}_{p,Q}(\pi) \boxdot_{p,Q} \delta$
  else
    $\mathsf{idx} = (\delta'.e + 14 \bmod 8) || (\mathsf{TR}(\delta'.m, q-2) \bmod 4)$
    $(\theta_1, \theta_3) = L_{\mathsf{tan\text{-}coeffs}}(\mathsf{idx}), \mathsf{idx} \in \{0,1\}^5$
    $\mu = ((\theta_3 \boxdot_{p,Q} \delta \boxdot_{p,Q} \delta) \boxplus^*_{p,Q} \theta_1) \boxdot_{p,Q} \delta$
  $\nu = L_{\mathsf{tan\text{-}table}}(N), N \in \{0,1\}^7$
  $\zeta_1 = \mu \boxplus^*_{p,Q} \nu$; $\zeta_2 = \mathtt{Float}_{p,Q}(1) \boxminus^*_{p,Q} (\mu \boxdot_{p,Q} \nu)$
  $(\eta_1, \eta_2) = \mathbf{1}\{c = 1\}$? $(\zeta_2, \zeta_1) : (\zeta_1, \zeta_2)$
  $\gamma = \eta_1 \boxslash^*_{p,Q} \eta_2$
  Return $(\gamma.z, b \oplus \alpha.s, \mathcal{F}^{p,q,Q}_{\mathsf{Round}^*}(\gamma.e, \gamma.m))$

Fig. 13: *Floating-point* $\mathsf{tan}\pi_{8,23}(\alpha)$.

C. Cosine

The functionality for cosine is given in Figure 14, which first handles the easy cases of input $\alpha \in (-\infty, \infty)$ in Steps 1-5 as follows:

- If the input $|\alpha| \geq 2^{23}$, $\mathsf{cos}\pi(\alpha) = 1$ if α is an even integer and 0 if α is an odd integer. Since all floats with exponent > 23 are even integers, we only set the output to -1 if $\alpha.e = 23$ and $\alpha.m \bmod 2 = 1$.
- If the input $|\alpha| < 2^{-14}$, it is small enough that approximating $\mathsf{cos}\pi(\alpha)$ as 1 has less than 1 ulp error.

For the rest of the inputs, we reduce the computation to evaluating sine, for which we already have a functionality. Specifically, using identity $\mathsf{cos}\pi(\alpha) = \mathsf{sin}\pi(\alpha + 0.5)$, we add 0.5 to the fixed-point integer m' representing α in Step 8, and then the remaining steps are the same as that in sine. Like sine, our cosine functionality also uses two splines F^1 and F^2 with identical configuration, but with different sets of coefficients Θ^1_{cos} and Θ^2_{cos}, respectively. Since $\mathsf{cos}\pi(\alpha) = \mathsf{cos}\pi(-\alpha)$, we do not flip the sign of the final output depending on the sign of α.

Functionality $\mathcal{F}^{8,23}_{\mathsf{FPcos}\pi}(\alpha)$

$p = 8$; $q = 23$; $Q = 27$; $(c_{\mathsf{LT}}, c_{\mathsf{EQ}}) = \mathsf{LT\&EQ}(23, \alpha.e)$
if $c_{\mathsf{LT}} \oplus c_{\mathsf{EQ}}$ **then**
  Return $(0, (\alpha.m \bmod 2) \wedge c_{\mathsf{EQ}}, 0, 2^q)$
else if $\mathbf{1}\{\alpha.e < -14\}$ **then**
  Return $\mathtt{Float}_{p,q}(1)$
else
  $m' = \alpha.m *_{q+15} 2^{\alpha.e+14}$
  $m = m' + 2^{q+13} \bmod 2^{q+15}$
  Perform Steps 10 - 28 on m from Figure 7 to get μ, a
  Return $(\mu.z, a, \mathcal{F}^{p,q,Q}_{\mathsf{Round}^*}(\mu.e, \mu.m))$.

Fig. 14: *Floating-point* $\mathsf{cos}\pi_{8,23}(\alpha)$.

Input	Output		ULP Error
	ABY-F [27]	Ideal/SECFLOAT	
SIN			
$7.5891 \cdot 10^{-8}$ `0x33a2f983`	0	$2.38419 \cdot 10^{-7}$	$16.7 \cdot 10^6$
COS			
0.49999 `0x3effffff`	$3.5 \cdot 10^{-7}$	$9.36268 \cdot 10^{-8}$	$37.1 \cdot 10^6$
TAN			
$1.89727 \cdot 10^{-8}$ `0x32a2f983`	0	$5.96046 \cdot 10^{-8}$	$16.7 \cdot 10^6$
LOG2			
0.99999 `0x3f7fffff`	$-1.1 \cdot 10^{-7}$	$-8.59913 \cdot 10^{-8}$	$4.6 \cdot 10^6$

Table V: Inputs which lead to large ULP errors in ABY-F circuits. Ideal is the exact real result. The floating-point inputs are shown both in their decimal as well as hexadecimal representations.

D. Example inputs with imprecise outputs in ABY-F

In Table V, we provide example inputs that lead to very imprecise outputs in ABY-F circuits [27]. We did not evaluate the precision of these libraries exhaustively; these are the inputs that have the maximum error in the tests we ran and larger errors are possible. The third column shows the exact real result with six decimal digits and SECFLOAT produces the same answer. The imprecision in ABY-F's sine is particularly worrisome as when it is used in computing cosecant ($\frac{1}{\sin}$) or cotangent ($\frac{\cos}{\sin}$) then the user will run into divide by zero errors on perfectly valid inputs. For reference, Intel recalled its pentium processors when DIV(4195835,3145727) returned 1.33373 instead of 1.33382. In Table V, even the exponent of the ABY-F output is wrong.

E. Performance numbers of ABY

In this section, we provide performance numbers of ABY for both garbled circuits (GC) and GMW. See Table VI for microbenchmark numbers.

F. Silent OT and extensions.

Another baseline we consider is running accurate functionalities with GMW [34] where the required oblivious transfers (OTs) are preprocessed using SilentOT extension [18], [26] and pseudorandom correlation generators (PCGs) [72]. We ran Ferret [72], which is the state-of-the-art, to estimate the cost of

Operation		Time (in s) for Batch Size			Comm. (KiB)	ULP Error
		10^3	10^4	10^5		
CMP	GC	0.11 (5.5x)	0.71 (5.6x)	6.94 (23.2x)	8.82 (7.9x)	NA
	GMW	0.07 (3.2x)	0.31 (2.4x)	2.81 (9.4x)	6.93 (6.2x)	
MULT	GC	0.79 (16.0x)	8.45 (60.4x)	83.74 (121.4x)	96.26 (30.7x)	0.5
	GMW	0.41 (8.3x)	3.89 (27.8x)	36.34 (52.7x)	95.74 (30.6x)	
ADD	GC	0.52 (3.4x)	5.13 (9.2x)	52.17 (13.3x)	60.04 (5.4x)	0.5
	GMW	0.29 (1.9x)	2.34 (4.2x)	25.12 (6.4x)	58.96 (5.3x)	
DIV	GC	1.44 (10.0x)	15.15 (32.8x)	–	170.61 (11.5x)	0.5
	GMW	0.80 (5.5x)	6.74 (14.6x)	63.33 (22.1x)	171.25 (11.5x)	
SIN	GC	1.37 (3.0x)	14.36 (9.0x)	–	164.98 (3.9x)	$> 10^7$
	GMW	0.72 (1.6x)	7.13 (4.5x)	62.29 (5.5x)	165.54 (3.9x)	
COS	GC	1.36 (2.5x)	14.17 (8.9x)	–	165.36 (3.9x)	$> 10^7$
	GMW	0.75 (1.4x)	6.41 (4.0x)	62.73 (5.6x)	165.92 (3.9x)	
EXP2	GC	2.63 (6.3x)	24.97 (16.2x)	–	306.39 (8.2x)	1
	GMW	1.26 (3.0x)	13.32 (8.6x)	112.83 (10.0x)	309.15 (8.3x)	
LOG2	GC	2.89 (5.5x)	27.84 (13.8x)	–	332.26 (6.5x)	$> 10^6$
	GMW	1.38 (2.6x)	12.63 (6.3x)	123.44 (8.3x)	335.44 (6.5x)	

Table VI: Performance numbers for ABY-GC and ABY-GMW. The numbers in parentheses show SECFLOAT's improvement factor. '-' denotes tool crash.

OT preprocessing for division from ABY-F and OpenLibm's sine, and observed that the time for this preprocessing alone[8] was $16.4\times$ and $1152\times$ worse than the *total* runtime of SECFLOAT. The communication of Ferret+GMW for division is $2.2\times$ better and for sine is $36\times$ worse than SECFLOAT. A recent work Silver [26] improves upon Ferret using new non-standard assumptions. Using Silver, preprocessing time is $4.5\times$ and $315\times$ worse than total runtime of SECFLOAT for division and sine respectively. The communication of Silver+GMW for division is $3.8\times$ better and for sine is $21\times$ worse than SECFLOAT.

Above comparisons used traditional IKNP-style [40] OT extensions for building blocks of SECFLOAT. We note that almost all of SECFLOAT's building blocks in Section IV-D use OTs and can benefit from SilentOT/PCG techniques, that trade communication with compute, to further improve SECFLOAT's performance in low bandwidth settings. In our evaluation set up, IKNP works better for SECFLOAT.

G. 2PC Building Blocks

We discuss the building blocks that prior work provides and the new building blocks that SECFLOAT requires.

[8]Ignoring the runtime of online multi-round GMW.

1) Existing building blocks:

a) Zero extension.: For $x \in \mathbb{Z}_{2^m}$, zero-extension to $n > m$ bits is defined as $\mathsf{ZXt}(x,n) = \zeta_m(x) \bmod 2^n$. [63] provides a protocol $\Pi^{m,n}_{\mathsf{ZXt}}$ that takes as input $\langle x\rangle^m$ and outputs $\langle \mathsf{ZXt}(x,n)\rangle^n$ with communication of at most $\lambda(m+1)+13m+n$ bits. When $m=1$, we do extension using the cheaper Boolean-to-Arithmetic (B2A) protocol, which costs $\lambda+n$ bits [64].

b) Truncate-and-Reduce.: We denote truncate-reduce by s bits as $\mathsf{TR}(x,s)$ that takes x of ℓ bits, drops the lower s bits and returns upper $\ell-s$ bits as the truncated output. [63] provides a protocol $\Pi^{\ell,s}_{\mathsf{TR}}$ that takes $\langle x\rangle^\ell$ as input and returns $\langle \mathsf{TR}(x,s)\rangle^{\ell-s}$ with communication $\lambda(s+1)+\ell+13s$ bits.

c) Multiplication with non-uniform bitwidths.: Unsigned multiplication between $x \in \mathbb{Z}_{2^m}$ and $y \in \mathbb{Z}_{2^n}$ with the output in an ℓ-bit ring is denoted by $x *_\ell y = \zeta_m(x)\cdot\zeta_n(y) \bmod 2^\ell$. [63] gives a protocol $\Pi^{m,n,\ell}_{\mathsf{UMult}}$ that takes $\langle x\rangle^m$ and $\langle y\rangle^n$ as input and returns $\langle x *_\ell y\rangle^\ell$. The communication complexity of this protocol is roughly $\lambda(3\mu+\nu)+\mu(\mu+2\nu)+16(m+n)$ where $\mu = \mathsf{min}(m,n)$ and $\nu = \mathsf{max}(m,n)$. Signed multiplication is denoted by $x *'_\ell y = \mathsf{int}_m(x)\cdot\mathsf{int}_n(y) \bmod 2^\ell$, and its protocol $\Pi^{m,n,\ell}_{\mathsf{SMult}}$ has the same communication as the unsigned case.

d) MSNZB: For an ℓ-bit number x, $\mathsf{MSNZB}(x)$ is defined as the index of the most significant non-zero bit. That is, $\mathsf{MSNZB}(x) = k \in \{0,\ldots,\ell-1\}$ such that $x_k = 1$ and $x_i = 0$ for all $i > k$. We make slight modifications[9] to protocol from [63] to create a protocol $\Pi^{\ell}_{\mathsf{MSNZB}}$ that takes $\langle x\rangle^\ell$ as input and outputs $\{\langle z_i\rangle^\ell\}_{i\in[\ell]}$ such that $z_i = 1$ for $i = \mathsf{MSNZB}(x)$ and 0 otherwise. Given $\{\langle z_i\rangle^\ell\}$, the parties can locally compute $\langle k\rangle^\ell = \sum_{i\in[\ell]} i\cdot\langle z_i\rangle^\ell$ and $\langle K\rangle^\ell = \sum_{i\in[\ell]} 2^{\ell-1-i}\cdot\langle z_i\rangle^\ell$. Alternatively, garbled circuits (GC) can be used to get boolean shares of one-hot encoding of MSNZB using $2(\ell-1)$ AND operations and ℓ B2A operations, which costs $\lambda(5\ell-4)+\ell^2$ bits. The MSNZB protocol from [63] is more communication-efficient than the GC solution; please refer [63] for the exact cost expression.

e) Comparison.: We require a protocol that takes $\langle x\rangle^\ell$ and $\langle y\rangle^\ell$ as input and returns $\langle \mathbf{1}\{x<y\}\rangle^B$ as output. Specialized protocols for this function instead compute $\mathbf{1}\{x-y<0\}$ and hence, for correctness require that $|x-y| < 2^{\ell-1}$ for unsigned inputs and $|x|+|y| < 2^{\ell-1}$ for signed inputs. Due to this constraint, while invoking the comparison protocol for greater-than Π^{ℓ}_{GT}, we set the bitwidths of x,y such that this condition is met. We use the protocol for comparison from [64] that has communication of $< \lambda\ell + 14\ell$ bits (for both signed and unsigned numbers).

f) Equality.: We use a slightly modified version of the millionaires' protocol from [64] to compute equality-check. In particular, their millionaires' protocol already does most of the work required for equality-check, and we can get an equality-check protocol Π^{ℓ}_{EQ} straightforwardly from it by avoiding the extra AND computations. The communication cost of the resultant protocol is $< \frac{3}{4}\lambda\ell + 9\ell$ bits.

[9] [63] produces boolean shares of one-hot encoding. To get arithmetic shares, we append it with COT-based boolean-to-arithmetic share conversion from [64].

g) Lookup Tables.: The lookup table (LUT) protocol $\Pi^{m,n}_{\mathsf{LUT}}$ takes a table L with 2^m entries of n-bits each and $\langle x\rangle^m$ as inputs, and outputs $\langle z\rangle^n$ such that $z = L[x]$. A single call to $\binom{2^m}{1}$-OT_n realizes this functionality with communication $2\lambda + 2^m n$ bits [29].

2) New Building Blocks: Our new building blocks, namely, less-than-and-equal (LT&EQ) and truncate-and-reduce-with-sticky-bit (TRS), reduce to a functionality called wrap-and-all-zeros $\mathsf{Wrap\&All0s}(\langle x\rangle^\ell)$. This functionality on $\langle x\rangle^\ell$ outputs two secret-shared bits $\langle w\rangle^B$ and $\langle z\rangle^B$, where wrap $w = \mathbf{1}\{\langle x\rangle^\ell_0 + \langle x\rangle^\ell_1 \geq 2^\ell\}$, and all-zero result $z = \mathbf{1}\{\langle x\rangle^\ell_0 + \langle x\rangle^\ell_1 = 0 \bmod 2^\ell\}$. The reductions of LT&EQ and TRS to Wrap&All0s are as follows:

- LT&EQ: Since $|x|+|y| < 2^{\ell-1}$, we can express the 2 output bits as $c_{\mathsf{LT}} = \mathbf{1}\{c < 0\}$ and $c_{\mathsf{EQ}} = \mathbf{1}\{c = 0\}$, respectively, where $c = x - y$. From [64], $c_{\mathsf{LT}} = \mathsf{MSB}(c) = \mathsf{MSB}(\langle c\rangle^\ell_0) \oplus \mathsf{MSB}(\langle c\rangle^\ell_1) \oplus d_w$, where $d_w = \mathbf{1}\{d_0 + d_1 \geq 2^{\ell-1}\}$, with $d_i = \langle c\rangle^\ell_i \bmod 2^{\ell-1}$. Similarly, $c_{\mathsf{EQ}} = \neg\mathsf{MSB}(c) \wedge d_z = \neg c_{\mathsf{LT}} \wedge d_z$, where $d_z = \mathbf{1}\{d_0 + d_1 = 0 \bmod 2^{\ell-1}\}$. Now, observe that Wrap&All0s on $\langle d\rangle^{\ell-1}$ outputs precisely (d_w, d_z). Hence LT&EQ can be computed by a single call to Wrap&All0s and a single AND.
- TRS: For $i \in \{0,1\}$, let $x_{\ell-s,i} \in \{0,1\}^{\ell-s}$ and $x_{s,i} \in \{0,1\}^s$ be the upper $\ell - s$ and lower s bits of $\langle x\rangle^\ell_i$, respectively. Now, $\mathsf{TRS}(x,s) = \mathsf{TR}(x,s) \vee \neg t_z$, where $t_z = \mathbf{1}\{x_{s,0} + x_{s,1} = 0 \bmod 2^s\}$. From [63], $\mathsf{TR}(x,s) = x_{\ell-s,0} + x_{\ell-s,1} + \mathsf{ZXt}(t_w, \ell - s)$, where $t_w = \mathbf{1}\{x_{s,0} + x_{s,1} \geq 2^s\}$. Wrap&All0s on $\langle x_s\rangle^s$ outputs t_w and t_z and from this TRS can be computes using a 1 OR and 1 zero-extension.

Finally, to compute Wrap&All0s, we use ideas similar to what was used in [63] to compute $\mathsf{Mill\&Eq}(x,y)$. This functionality takes $\langle x\rangle^\ell, \langle y\rangle^\ell$ as input and outputs $\langle c\rangle^B = \mathbf{1}\{x > y\}$ and $\langle e\rangle^B = \mathbf{1}\{x = y\}$ and could be computed at nearly the same communication cost as computing only $\langle c\rangle^B$ ($\lambda\ell + 14\ell$ bits). By writing w and z in Wrap&All0s as $w = \mathbf{1}\{x_0 > 2^\ell - x_1\} \oplus \mathbf{1}\{x_0 = 2^\ell - x_1\}$ and $z = \mathbf{1}\{x_0 = 2^\ell - x_1\} \wedge \mathbf{1}\{x_0 = x_1 = 0\}$, the values $a = \mathbf{1}\{x_0 > 2^\ell - x_1\}$ and $b = \mathbf{1}\{x_0 = 2^\ell - x_1\}$ can be computed using Mill&Eq on $(\ell+1)-$bit inputs. Now, $w = a \oplus b$ and $z = b \wedge \mathbf{1}\{x_0 = 0\} \wedge \mathbf{1}\{x_1 = 0\}$. Hence, Wrap&All0s can be computed with a single call to Mill&Eq and 2 AND operations.

H. Floating-Point Comparison $<_{p,q}$

In Figure 15, given two floating-point values α_1 and α_2, to check if $\alpha_1 < \alpha_2$, we start by comparing their signs. If the signs are unequal then we return the sign of α_1, except if α_1 and α_2 are both ± 0, in which case we return false. If the signs are equal and positive (the negative case is similar) then we compare the exponents and return true if exponent of α_1 is smaller. If the exponents are equal as well then we compare the mantissa and return true if the mantissa of α_1 is smaller. If the sign, exponent, and mantissa of α_1 and α_2 are equal then we return false.

Functionality $\mathcal{F}^{p,q}_{\mathsf{FPLT}}(\alpha_1, \alpha_2)$

$s_{\mathsf{EQ}} = 1 \oplus \alpha_1.s \oplus \alpha_2.s$
$(e_{\mathsf{LT}}, e_{\mathsf{EQ}}) = \mathsf{LT\&EQ}(\alpha_1.e, \alpha_2.e)$
$(m_{\mathsf{LT}}, m_{\mathsf{EQ}}) = \mathsf{LT\&EQ}(\alpha_1.m, \alpha_2.m)$
if $\mathbf{1}\{s_{\mathsf{EQ}} = 1\}$ **then**
    Return $(\alpha_1.s \oplus e_{\mathsf{LT}} \oplus (e_{\mathsf{EQ}} \wedge m_{\mathsf{LT}})) \wedge \neg(m_{\mathsf{EQ}} \wedge e_{\mathsf{EQ}})$
else
    Return $\alpha_1.s \wedge \neg(\alpha_1.z \wedge \alpha_2.z)$

Fig. 15: *Floating-Point Less Than:* $\alpha_1 <_{p,q} \alpha_2$

Functionality $\mathcal{F}^{p,q}_{\mathsf{CheapFPAdd}}(\alpha_1, \alpha_2)$

$(e_{\mathsf{LT}}, e_{\mathsf{EQ}}) = \mathsf{LT\&EQ}(\alpha_1.e, \alpha_2.e)$
$m_{\mathsf{LT}} = \mathbf{1}\{\alpha_1.m < \alpha_2.m\}$
$(\beta_1, \beta_2) = e_{\mathsf{LT}} \oplus (e_{\mathsf{EQ}} \wedge m_{\mathsf{LT}})\ ?\ (\alpha_2, \alpha_1) : (\alpha_1, \alpha_2)$
$d = \beta_1.e - \beta_2.e$
if $\mathbf{1}\{d > q + 1\}$ **then**
    Return β_1
else
    $m_1 = \mathsf{ZXt}(\beta_1.m, q+2)$
    $m_2 = \mathsf{TR}(\beta_2.m *_{2q+3} 2^{q+1-d}, q+1)$
    $m_2 = (\beta_1.s \oplus \beta_2.s\ ?\ -1 *_{q+2} m_2 : m_2)$
    $m = m_1 + m_2$; $e = \beta_2.e$
    $k, K = \mathsf{MSNZB}(m)$
    $m = \mathsf{TR}(m *_{q+2} K, 1)$; $e = e + k - q$
    $z = \mathbf{1}\{m = 0\}$; $s = \beta_1.s$
    Return $\alpha = (z, s, e, m)$

Fig. 16: *Cheaper Floating-Point Addition:* $\alpha_1 \boxplus^*_{p,q} \alpha_2$

I. Optimizations

1) Floating-point Multiplication: In our floating-point multiplication functionality (Figure 4), we can use the TRS operation on m in Step 2 to discard its lower $q-2$ bits and work with smaller bitwidths in the subsequent operations. Specifically, the comparison in Step 3 changes to $m < 2^{q+3} - 2$ in $q+4$ bits, and the rounding in Step 4 and Step 6 is done by 2 and 3 bits, respectively. The main benefit of this optimization is the cheaper comparison in Step 3. Since $(\mathsf{TRS}(x, s-k)) \gg_R k = x \gg_R s$, where $s - k \geq 2$, this optimization does not change the behaviour of the above rounding operations, and consequently, the output of the functionality.

2) Cheap Floating-Point Addition: In Figure 16, we provide the cheap variant of our floating-point addition functionality which is imprecise. Apart from using truncate-reduce in place of rounding, it differs from our correct addition functionality (Figure 5) in one major way: to align the mantissas of the operands, instead of left-shifting the larger operand (in magnitude) by the difference of exponents, we right-shift the smaller operand by the same amount. As a result, we are able to work with nearly half the bitwidth in all subsequent steps, while losing precision due to significant bits being discarded. Concretely, this functionality has $1.57\times$ and $1.63\times$ less communication than Figure 5 for $p = 8, q = 23$ and $p = 8, q = 27$, respectively.

3) Cheap Floating-Point Division: Our cheap floating-point division functionality is given in Figure 17. It is the same as our correct division functionality (Figure 6), except it returns

Functionality $\mathcal{F}^{p,q}_{\mathsf{CheapFPDiv}}(\alpha_1, \alpha_2)$

1: Perform Steps 1 – 11 from Figure 6 to get e and m'
2: $s = \alpha_1.s \oplus \alpha_2.s$, $z = \alpha_1.z$
3: Return $\alpha = \mathcal{F}^{p,q}_{\mathsf{FPCheck}}(z, s, e, m')$

Fig. 17: *Cheaper Floating-point Division:* $\alpha_1 \boxslash^*_{p,q} \alpha_2$

$\mathcal{F}^{\ell,n,d,p,q}_{\mathsf{GetC}}(x, \Theta = \{\theta_i^{(j)}\}_{i=0,j=1}^{d,n}, K = \{k_j\}_{j=1}^{n+1})$

1: $\{v_j\}_{j\in[n]} = L^K_{\mathsf{ActiveInterval}}(x), x \in \{0,1\}^\ell, v_j \in \{0,1\}$
2: For $j \in [n]$, $V_j = \mathsf{ZXt}(v_j, q+1)$; $V'_j = V_j \bmod 2^{p+2}$
3: **for** $i = 0$ to d **do**
4: $\quad z_i = s_i = e_i = m_i = 0$
5: $\quad$ **for** $j = 1$ to n **do**
6: $\quad\quad z_i = z_i + v_j \cdot \theta_i^{(j)}.z$; $s_i = s_i + v_j \cdot \theta_i^{(j)}.s$
7: $\quad\quad e_i = e_i + V'_j \cdot \theta_i^{(j)}.e$; $m_i = m_i + V_j \cdot \theta_i^{(j)}.m$
8: Return $\{(z_i, s_i, e_i, m_i)\}_{i=0}^d$

Fig. 18: *Retrieve the coefficients of the active piece in an n-piece d-degree spline using an ℓ-bit index x:* $\mathsf{GetC}_{\ell,n,d,p,q}(x, \Theta, K)$.

m' as the mantissa which has at most 1 ULP error. Using this functionality leads to $1.22\times$ less communication than Figure 6.

J. Spline implementation

There are two natural ways to realize splines. 1) Compare idx with $\{k_1, \ldots, k_{n+1}\}$ and depending on the comparison outputs, compute the correct coefficients [57], [58]. 2) Use a 2^ℓ size LUT that maps idx to correct coefficients. Both of these approaches have high cryptographic cost. The former requires n comparisons. In the latter approach, the lookup table entries are $d+1$ floating-point numbers, and since the cost of lookup table grows linearly with entry size, this approach has high cost when we are looking for many coefficients. We give a specialized crypto-friendly functionality for selecting the correct coefficients in Figure 18.

First, we construct a LUT $L^K_{\mathsf{ActiveInterval}}$ of size 2^ℓ (of one-hot encodings) that maps $\mathsf{idx} \in \{0,1\}^\ell$ to $v \in \{0,1\}^n$ such that $v_j = 1$ if $\mathsf{idx} \in [k_j, k_{j+1})$ and 0 otherwise. Note that the table entries are only n-bits. After the lookup in Step 1, we compute the correct coefficients as $\sum_j v_j \cdot \theta_j$. To compute this with our floating-point representation, we extend v_j to appropriate size rings (Step 2) before multiplication. We crucially note that since the coefficients in Θ are public, the above equation can be computed locally by each party using scalar multiplication on secret shares of v_j. Hence, our cryptographic cost is only proportional to 2^ℓ-size LUT with n-bit entries followed by n extensions from 1 bit to $q+1$ bits. That is, our cryptographic cost is independent of the degree of the polynomial used in the spline[10]. Moreover, we carefully choose the knots to ensure that ℓ is small. All our splines have $\ell \leq 8$ and $n \leq 64$.

[10]Number of multiplications on δ needed to compute the polynomial depends on the degree, and hence, we aim to minimize the degree.

Concretely, for tangent that requires only 2 coefficients, the naïve LUT-based approach is better. But for functions like exponentiation and logarithm (Section VI-C) that require looking up 6 and 8 coefficients respectively, our approach has $2.3\times$ and $3.7\times$ lower communication than naïve LUT-based approach, respectively. Also, our approach has $3.1\times$ and $3.9\times$ lower communication than comparison-based approach for exponentiation and logarithm, respectively.

K. Generating splines.

We use *splines*, or piecewise polynomials for polynomial approximations for elementary math functions f. As discussed in Section VI-A, an n-piece spline is characterized by *configurations* $c = (\mathcal{K}, d)$, consisting of *knots* $\mathcal{K}$ and a polynomial *degree* d, and the polynomial coefficients $\Theta = \{\theta_i^{(j)}\}_{i=0,j=1}^{d,n}$. We decompose the problem to determine splines into generating polynomial coefficients Θ (given a configuration) and generating a configuration ($\mathcal{K}$, d):

1) Generating coefficients given a configuration: For a given configuration, RLIBM [51] provides a mechanism to generate Θ from three inputs, $f(\alpha)$, range reduction function and output compensation, using linear programming and CEGIS (counterexample guided inductive synthesis). RLIBM uses it to generate polynomials that when evaluated with double precision floating-point give correctly rounded single-precision floating-point results after output compensation. We generalize RLIBM's implementation in two aspects to obtain our coefficient generator. First, instead of being limited to correctly rounded results, we create linear programs whose solutions are polynomial coefficients which ensure that the final result is within 1 ULP of $f(\alpha)$. Second, instead of limiting to polynomials over doubles, we provide general support for evaluating polynomials using floating-point numbers with parameters (p, q). This generalized implementation determines the coefficients for a given configuration automatically if they exist and fails otherwise.

2) Generating configurations: To generate an appropriate polynomial approximation, SECFLOAT (with manual guidance) explores candidate configurations in increasing cryptographic cost and runs the coefficient generator for each such configuration. The cryptographic cost crucially depends on the degree d of the polynomials and number of bits ℓ of the input that determine the active interval and hence the configurations are explored in increasing d, ℓ. (Note that the cryptographic cost of a spline is entirely governed by its configuration and is independent of the actual polynomial coefficients.) The exploration stops when the coefficient generator succeeds.

L. Knots for Math Functions

In this section, we provide details on the knots used in splines for our math implementations. The polynomial coefficients can be obtained from the source code of SECFLOAT.

1) Sin/Cosine: Sine has two splines F^1 and F^2 which cover the input range $[2^{-14}, \frac{1}{32})$ and $[\frac{1}{32}, 0.5]$, respectively. F^1 has knots at powers of 2: $\kappa_{j+1} = 2^{j-14}$, $j \in [0, 9]$. The knots in F^2 are at multiples of $\frac{1}{32}$ in the range $[\frac{1}{32}, \frac{9}{32})$, at multiples

Technique	Time (in s) for Batch Size			Comm. (KiB)	Average ULP Error
	10^3	10^4	10^5		
Scenario I: Parties have secret inputs					
SECFLOAT	0.96	2.86	21.42	75.27	33.86
ABY-F	2.29 (2.4x)	24.33 (8.5x)	250.68 (11.7x)	833.98 (11.1x)	
Scenario II: Parties have secret-shares of inputs					
SECFLOAT	2.41	7.59	55.55	211.26	1.05
ABY-F	4.21 (1.7x)	40.27 (5.3x)	415.61 (7.5x)	1437.73 (6.8x)	19612.95

Table VII: Comparison of SECFLOAT with ABY-F on privacy-preserving proximity testing. The numbers in parentheses show our improvement factor.

of $\frac{1}{64}$ in the range $[\frac{9}{32}, \frac{10}{32})$, and lastly at multiples of $\frac{1}{128}$ in the remaining input range $[\frac{10}{32}, 0.5]$. The last interval in F^2 is closed on the right to include 0.5. The splines used in cosine have configuration identical to sine's splines, and thus, use the same sets of knots.

2) Tangent: Tangent uses a single spline for the input range $[2^{-14}, \frac{1}{512}]$ and has knots at $\{2^j, 1.25 \cdot 2^j, 1.5 \cdot 2^j, 1.75 \cdot 2^j\}$, for $j \in [-14, -10]$. Apart from these, there are two more knots at $\frac{1}{512}$ and $1.25 \cdot \frac{1}{512}$. Note that the final piece has just one possible input.

3) Exponentiation: Exponentiation has two splines F^+ and F^- that share the same configuration. They handle the input range $[2^{-24}, 1)$, and have all knots at multiples of $\frac{1}{64}$ in the range $[\frac{1}{64}, 1]$, except the first one which is at 2^{-24}.

4) Logarithm: Logarithm uses four splines F^1, F^2, F^3, and F^4. F^1 and F^3 handle the input range $[2^{-24}, \frac{1}{32})$ and $[2^{-23}, \frac{1}{32})$, respectively, and both have knots at powers of 2. The input domain for F^2 is $[\frac{1}{32}, 0.5]$ and it has knots at multiples of $\frac{1}{32}$ in $[\frac{2}{32}, \frac{13}{32})$ and at multiples of $\frac{1}{64}$ in $[\frac{1}{32}, \frac{2}{32}) \cup [\frac{13}{32}, \frac{33}{64}]$. Note that the final piece has just one possible input, i.e., 0.5. Finally, F^4 covers the input range $[\frac{1}{32}, 1)$ and has knots at multiples of $\frac{1}{32}$ in $[\frac{2}{32}, 1]$, along with two more knots at $\{\frac{1}{32}, \frac{3}{64}\}$.

M. Privacy-Preserving Proximity Testing

We evaluate SECFLOAT on privacy preserving proximity testing [70], the case study used in ABY-F [27]. Here, given secret coordinates (ψ_A, ϕ_A) and (ψ_B, ϕ_B) for points A and B on earth, where ψ_A, ψ_B are latitudes and ϕ_A, ϕ_B are longitudes, it tests whether the distance Δ between the points is less than a threshold ϵ. The distance Δ is calculated using Haversine's formula [68]:

$\Delta = 2R \cdot \mathsf{tan}^{-1}(\sqrt{\frac{\delta}{1-\delta}})$, where R is the radius of the Earth, $\delta = \mathsf{sin}\pi^2(\frac{\psi_A - \psi_B}{2}) + \mathsf{cos}\pi(\psi_A) \cdot \mathsf{cos}\pi(\psi_B) \cdot \mathsf{sin}\pi^2(\frac{\phi_A - \phi_B}{2})$.

The secure computation of $\Delta < \epsilon$ can be reduced to securely computing δ [70]. The last column of Table VII shows the ULP error in computing δ in two scenarios.

In **scenario I**, Alice knows A and Bob knows B and they want to securely compute whether distance between A and B is less than ϵ. As shown by [27], [70], the computation can be rearranged using trigonometric identities that allows local computation of trigonometric functions and we need to only compute the following floating-point operations securely: 6 multiplications, 5 additions, and 1 comparison. We run this computation with both ABY-F circuits [27] and SECFLOAT and report the results in the top half of Table VII (we take the best number of ABY and EMP). The primitive operators of both SECFLOAT and ABY-F are correct and result in similar numerical errors. However, SECFLOAT's communication is an order of magnitude lower.

In **scenario II**, Alice and Bob both have secret shares of (ψ_A, ϕ_A) and (ψ_B, ϕ_B), and want to compute secret shares of $\Delta < \epsilon$. This scenario arises when privacy preserving proximity testing is a part of a bigger computation in which the coordinates are the output of some 2PC and cannot be revealed to any single party. This computation requires 4 multiplications, 3 additions, 2 divisions, 1 comparison, 2 $\mathsf{sin}\pi$ and 2 $\mathsf{cos}\pi$ operations. Here, the trigonometric computations need to be performed with 2PC and the output of ABY-F is very imprecise (the bottom half of Table VII). On 10^5 randomly generated pairs of coordinates, the average ULP error of [27] is 19613 ULPs, which means that more than half of the mantissa bits in the output are garbage on average. In contrast, the average numerical error of SECFLOAT is ≈ 1 ULP. In meters, this translates to ABY-F having an average deviation of $\approx$ 2 meters, while SECFLOAT has an average deviation of $\approx 10^{-3}$ meters with $6.8\times$ lower communication.

We conclude that SECFLOAT provides an accurate implementation of this application. In both the scenarios, the ULP error of SECFLOAT-based implementation matches the error of the cleartext implementation that runs the floating-point computation with Intel's compilers and libraries. Moreover, the communication of SECFLOAT are $6.8 - 11\times$ lower than the prior state-of-the-art 2PC implementations of this application and provide up to an order of magnitude faster runtimes.

N. ReLU Protocol

Rectified Linear Unit (or ReLU) is defined as $\mathsf{ReLU}(\alpha) = (\alpha < 0 \ ? \ 0 : \alpha)$. Thus, ReLU can realized very simply using the multiplexer (MUX) protocol: $\Pi_{\mathsf{MUX}}^{p+q+5}(\langle \alpha.s \rangle^B, \langle 0 \rangle^{\mathsf{FP}(p,q)}, \langle \alpha \rangle^{\mathsf{FP}(p,q)})$. The naïve method of performing a MUX on a secret-shared floating-point value is to do an independent MUX on each of its component. This results in 4 MUX operations and a communication cost of $8\lambda + 2(p + q + 5)$ bits using the MUX protocol from [64][11].

We optimize this operation with the simple observation that all MUXes are using the shares of the same selection bit. Thus, these MUXes can be combined using the COT-batching trick from [58] into a single MUX on $p + q + 5$ bits, leading to a communication cost of just $2(\lambda + p + q + 5)$ bits.

[11] One bit each from zero and sign bits, $p+2$ bits from exponent, and $q+1$ bits from mantissa