Secure and Accurate Summation of Many Floating-Point Numbers

Marina Blanton
University at Buffalo
Buffalo, NY, USA
mblanton@buffalo.edu

Michael T. Goodrich
University of California, Irvine
Irvine, CA, USA
goodrich@acm.org

Chen Yuan[*]
Meta Platform Inc.
Bellevue, WA, USA
chenyc@meta.com

ABSTRACT

Motivated by the importance of floating-point computations, we study the problem of securely and accurately summing many floating-point numbers. Prior work has focused on security absent accuracy or accuracy absent security, whereas our approach achieves both of them. Specifically, we show how to implement floating-point superaccumulators using secure multi-party computation techniques, so that a number of participants holding secret shares of floating-point numbers can accurately compute their sum while keeping the individual values private.

KEYWORDS

floating-point summation, superaccumulator, secret sharing

1 INTRODUCTION

Floating-point numbers are the most widely used data type for approximating real numbers with a wide variety of applications; see, e.g., [30, 41, 51]. A (radix-2) floating-point number x is a tuple of integers (b, v, p) such that

$$x = (-1)^b \times (1 + 2^{-m} v) \times 2^{p-2^{e-1}-1}, \tag{1}$$

where $b \in \{0, 1\}$ is a *sign bit*, v is the m-bit *mantissa* (which is also known as the *significand*), and p is the e-bit *exponent*.

A well-known issue with floating-point arithmetic is that it is not exact. For example, it is known that summing two floating point numbers can have a roundoff error and these roundoff errors can propagate and even become larger than a computed result when performing a sequence of many floating-point additions. For example, floating-point addition is not associative [37].

Floating-point arithmetic has applications in many areas including medicine, defense, economics, and physics simulation (e.g., in the NVIDIA Omniverse [33]). Thus, there is considerable need in computing sums of many floating-point numbers as accurately as possible. For example, the accuracy of any computation that involves high-dimensional dot products or matrix multiplications, such as in machine-learning (see, e.g., [26, 32]), depends on the accuracy of computing the sum of many floating-point numbers. Similarly, computations in computational geometry involve computing determinants, whose accuracy also depends on computing the sum of many floating-point numbers; see, e.g., [22, 45, 49].

In addition, the fact that floating-point addition is not associative presents problems related to the reproducibility of computations; see, e.g., [17, 21–23]. For example, a secure contract involving the summation of floating-point numbers may need to be verified after it has been signed. And if this depends on the summation of floating-point values, performing the summation on different computers could result in different outcomes, which could cause participants to reject an otherwise valid digital contract.

Competing with this issue is that some applications of floating-point arithmetic have computer-security requirements, including integrity, confidentiality, and privacy. For example, computing the probability of satellites colliding could involve security and privacy considerations when the satellites belong to competing companies or adversarial nation-states, e.g., see [36]. Thus, there is a need for protocols for computing sums of many floating-point numbers as securely as possible. This holds for other domains where computation on private data is performed using floating-point arithmetic including applications in medicine and privacy-preserving training of machine learning models on distributed sensitive data.

In spite of the importance of accuracy and security for summing floating-point numbers, we are not aware of any prior work that simultaneously achieves both accuracy and security for summing many floating-point numbers. As we review below, there is considerable prior work on methods for accurately summing many floating-point numbers, but the methods used do not lend themselves to transformations into secure computations. Likewise, as we also review below, there is considerable prior work on securely computing sums of pairs of floating-point numbers, but these prior methods do not consider the propagation of roundoff errors and can lead to inaccurate results for summing many floating-point numbers. Such inaccuracies can arise after adding numbers of significantly different magnitudes, where the values of the largest magnitude have opposite signs and significantly exceed other summation operands. Adding the values one at a time using floating-point addition can therefore leave us with noise, while implementing addition exactly will retain the necessary number of summation bits. Thus, in this paper, we are interested in methods for summing many floating-point numbers that are both secure and accurate.

Related Prior Work. Neal [42] describes algorithms using a number representation called a *superaccumulator* to exactly sum n floating point numbers, which is then converted to a faithfully-rounded floating-point number. Unfortunately, while Neal's superaccumulator representation reduces carry-bit propagation, it does not eliminate it, as is needed for the purposes of this work. A similar idea has been used in ExBLAS [17], an open source

[*] Majority of the work was performed while at the University at Buffalo.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–14

https://doi.org/XXXXXXX.XXXXXXX

library for floating point computations. Shewchuck [45] describes an alternative representation for exactly representing intermediate results of floating-point arithmetic, but the method also does not eliminate carry-bit propagation in summations; hence, it also does not satisfy our accuracy constraints. In addition to these solutions, there are a number of adaptive methods for exactly summing n floating point numbers using various other data structures for representing intermediate results, which do not consider the security or privacy of the data. Further, these methods, which include ExBLAS [17] and algorithms by Zhu and Hayes [52, 53], Demmel and Hida [21, 22], Rump *et al.* [46], Priest [43], Malcolm [39], Leuprecht and Oberaigner [38], Kadric *et al.* [35], and Demmel and Nguyen [23], are not amenable to conversion to secure protocols with few rounds.

While integer arithmetic in secure multi-party computation has been extensively investigated, secure floating-point arithmetic has only gradually attracted attention in the last decade. Catrina and Saxena [16] extended secure computation from integer pairwise arithmetic to fixed-point pairwise arithmetic and applied it to linear programming [15]. Franz and Katzenbeisser [28] proposed a solution, based on homomorphic encryption and garbled circuits, for floating-point pairwise operations in the two-party setting with no implementation or performance results. Aliasgari *et al.* [3] designed a set of protocols for basic floating-point operations based on Shamir secret sharing and developed several advanced operations such as logarithm, square root and exponentiation of floating-point numbers. Their solution was improved and extended for other settings and applications [2, 8, 36, 47] later. Dimitrov et al. [24] proposed two sets of protocols using new representations to improve efficiency, but did not follow the IEEE 754 standard representation. Archer et al. [6] measure performance of floating-point operations in different instantiations using a varying number of computation participants and corruption thresholds. Rathee et al. [44] design secure protocols in the two-party setting and exactly follow the IEEE standard rounding procedure. In addition to the above works on improving efficiency of unary/binary floating-point operations, Catrina [11–13] proposed and improved several multi-operand operations such as sum, dot-product, and polynomial evaluation. Nevertheless, because their solutions are still based on traditional floating-point pairwise addition, round-off errors accumulate inevitably in each addition operation.

Our Results. In this paper, we develop new secure protocols for summing many floating-point numbers that outperforms other approaches. We design a superaccumulator-based solution that privately and accurately calculates summations of many private arbitrary-precision floating-point numbers, and we empirically evaluate the performance of our solution on varying input sizes and precision. Unlike standard floating-point addition, our approach performs summation exactly without introducing round-off errors.

Our supperaccumulator-based approach and most of the protocols we develop can be instantiated with building blocks based on secret sharing in different settings, including computation with or without honest majority and semi-honest and malicious adversarial models. Some of the design choices are made in favor of reducing communication and one efficient low-level building block, conversion shares of a bit from binary to arithmetic sharing, is in the three-party setting with honest majority based on replicated secret sharing in the semi-honest model (as defined below). We implement the construction in that setting and show that its runtime is faster than the state of the art implementing floating-point operations [12, 44]. Thus, we are able to implement exact addition while simultaneously improving performance.

Algorithm 1 $s \leftarrow \mathsf{ExpandAndSum}(x_1, x_2, \ldots, x_n)$

```
for $i = 1, \ldots, n$ do
    $y_i \leftarrow \mathsf{ConvertToInt}(x_i)$;
end for
$v \leftarrow \sum_{i=1}^{n} y_i$; // exact addition
$s \leftarrow \mathsf{ConvertToFloat}(v)$;
return $s$;
```

2 FLOATING-POINT SUMMATION CONSTRUCTION

2.1 The Expand-and-Sum Solution

There is a simple naïve solution for exactly summing a set of n floating-point numbers, $\{x_1, x_2, \ldots, x_n\}$, which we refer to as the *expand-and-sum* solution. It is reasonable for low-precision floating-point representations and is given as Algorithm 1. That is, for each floating-point number x_i, we convert the representation of x_i into an integer y_i, with as many bits as is possible based on the floating-point type being used for the x_is. Then we sum these values exactly using integer addition and convert the result back into a floating-point number.

The y_is would have the following sizes based on the IEEE 754 formats:

- *Half*: a half-precision floating-point number in the IEEE 754 format has 1 sign bit, a 5-bit exponent, and a 10-bit mantissa. Thus, representing this as an integer requires $1 + 2^5 + 10 = 43$ bits.
- *Single*: a single-precision floating-point number has 1 sign bit, an 8-bit exponent, and a 23-bit mantissa. Thus, representing this as an integer requires $1 + 2^8 + 23 = 280$ bits.
- *Double*: a double-precision floating-point number has 1 sign bit, an 11-bit exponent, and a 52-bit mantissa. Thus, representing this as an integer requires $1 + 2^{11} + 52 = 2,101$ bits.
- *Quad*: a quad-precision floating-point number has 1 sign bit, a 15-bit exponent, and a 112-bit mantissa. Thus, representing this as an integer requires $1 + 2^{15} + 112 = 32,881$ bits.

Further, there are also even higher-precision floating-point representations, which would require even more bits to represent as fixed-precision or integer numbers; see, e.g., [10, 27, 29, 31, 50]. Implementing a summation using this representation would involve performing many operations on very large numbers using secure multi-party computation techniques, thus degrading performance. Of course, applications with high-precision floating-point numbers are likely to be applications that require accurate summations; hence, we desire solutions that can work efficiently for such applications without requiring ways of summing very large integers. In particular, summing very large integers requires techniques for dealing with cascading carry bits during the summations, and performing all these operations securely is challenging for very large integers. Thus, we consider this expand-and-sum approach for summing n floating-point numbers as integers to be limited to low-precision floating-point representations.

2.2 Superaccumulators

An alternative approach, which is better suited for use with conventional secure addition when applied to high-precision floating-point formats, is to use a *superaccumulator* to represent floating-point summands, e.g., see [17, 18, 42]. This approach also uses integer arithmetic, but with much smaller integers. More importantly, it can be implemented to avoid cascading carry-bit propagation.

In a superaccumulator, instead of representing a floating-point number as a single expanded (very-large) integer, we represent that integer as a sum of small components maintained separately. That is, we represent the expanded integer y, corresponding to a floating-point number x, as a vector of $2w$-bit integers $\langle y_\alpha, y_{\alpha-1}, \ldots, y_1 \rangle$, where $y = \sum_{i=1}^{\alpha} (2^w)^{i-1} y_i$ and $\alpha = \lceil \frac{2^e+m}{w} \rceil$, so that we cover all possible exponent values. Also, note that if we convert a floating-point number to a superaccumulator, then at most $\beta = \lceil \frac{m+1}{w} \rceil + 1$ of the entries will be non-zero. We can choose w based on the underlying mechanism for achieving security and privacy. For example, if we want to use built-in 64-bit integer addition, we can choose w to be 32.

In addition, we say that s is *regularized* if $-2^w < y_i < 2^w$ for $i = 1, \ldots, \alpha$. At a high level, in our scheme, we start with a regularized representation for each floating-point number x_i, and then we perform summations on an element-by-element basis. Finally, we regularize the partial sums by shifting "carry" values to neighboring elements. As we show, this approach allows us to prevent these carry values from propagating in a cascading fashion after performing a group of sums, which allows us to achieve efficiency for our secure summation protocols.

Suppose we are given n floating-point numbers, $\{x_1, x_2, \ldots, x_n\}$, each represented as a regularized superaccumulator $x_i = \sum_{j=1}^{\alpha} (2^w)^{j-1} y_{i,j}$. Further, suppose $n \le 2^{w-2}$. We sum all the x_i's by

- first summing the corresponding terms, $s_j = \sum_{i=1}^{n} y_{i,j}$,
- then splitting the binary representation of each s_j into c_{j+1} and r_j, so that $s_j = c_{j+1} 2^{w-1} + r_j$, where $-2^{w-1} < r_j < 2^{w-1}$,
- and lastly, updating each s_j as $s_j \leftarrow r_j + c_j$, for $j = 1, \ldots, n$.

As we show, because of the way that we regularize superaccumulators, the "carry" values, c_j, will not propagate in a cascading way, and the result of the above summation will be regularized. This allows us to complete the sum in a single communication round.

Further, for practical values of w, the constraint that $n \le 2^{w-2}$ is not restrictive. For example, if $w = 32$, this implies we can sum up to one billion floating-point numbers in a single communication round. Thus, to sum larger groups of numbers, we can group the summations in a tree where each internal node has 2^{w-2} children, and perform the sums in a bottom-up fashion. The important property, though, is that performing the above approach of summing $n \le 2^{w-2}$ regularized superaccumulators and then adding the carry values, c_j (some of which may be negative), to the neighboring element will result in a regularized superaccumulator. The following theorem establishes this property.

Theorem 2.1. *If $n \le 2^{w-2}$, then summing n regularized superaccumulators using the above algorithm will produce a regularized result.*

Proof. Let $x_1, x_2, \ldots, x_n$ be the set of input superaccumulators to sum, where $n \le 2^{w-2}$ and $x_i = \sum_{j=1}^{\alpha} (2^w)^{j-1} y_{i,j}$ for $i = 1, 2, \ldots, n$. Recall that we sum all the x_is by summing the corresponding terms, i.e., $s_j = \sum_{i=1}^{n} y_{i,j}$. Since each x_i is regularized, $-2^w < y_{i,j} < 2^w$ for all i, j. Thus, $-2^w n < s_j < 2^w n$ for all j; and hence, $-2^{2w-2} < s_j < 2^{2w-2}$ since $n \le 2^{w-2}$.

Recall that we split the binary representation of each s_j into c_{j+1} and r_j, so that $s_j = c_{j+1} 2^{w-1} + r_j$, where $-2^{w-1} < r_j < 2^{w-1}$. Thus,

$$s_j = c_{j+1} 2^{w-1} + r_j < c_{j+1} 2^{w-1} + 2^{w-1} = (c_{j+1} + 1) 2^{w-1} < 2^{2w-2}$$

and

$$s_j = c_{j+1} 2^{w-1} + r_j > c_{j+1} 2^{w-1} - 2^{w-1} = (c_{j+1} - 1) 2^{w-1} > -2^{2w-2}.$$

Therefore, $-2^{w-1} + 1 < c_{j+1} < 2^{w-1} - 1$ for each j. So, when we update each s_j as $s_j \leftarrow r_j + c_j$, then

$$s_j = r_j + c_j < 2^{w-1} + 2^{w-1} - 1 = 2^w - 1 \text{ and}$$
$$s_j = r_j + c_j > -2^{w-1} - 2^{w-1} + 1 = -2^w + 1.$$

Therefore, the result is regularized. □

3 SECURE COMPUTATION PRELIMINARIES

3.1 Security Setting

We use a conventional secure multi-party setting with N parties running the computation, t of which can be corrupt. Given a function f to be evaluated, the computational parties securely evaluate it on private data such that no information about the private inputs, or information derived from the private inputs, is revealed. More formally, a standard security definition requires that the view of the participants during the computation is indistinguishable from a simulated view generated without access to any private data.

Most of the protocols developed in this work can be instantiated in different adversarial models, but our implementation and one low-level building block are in the semi-honest model, in which the participating parties are expected to follow the computation, but might try to learn additional information from what they observe during the computation. Then the security requirement is that any coalition of at most t conspiring computational parties is unable to learn any information about private data that the computation handles. Achieving security in the semi-honest setting first is also important if one wants to have stronger security guarantees, and many of the protocols developed in this work would also be secure in the malicious model when instantiated with stronger building blocks.

Definition 3.1. Let parties $P_1, \ldots, P_N$ engage in a protocol Π that computes function $f(\mathsf{in}_1, \ldots, \mathsf{in}_N) = (\mathsf{out}_1, \ldots, \mathsf{out}_N)$, where in_i and out_i denote the input and output of party P_i, respectively. Let $\mathrm{VIEW}_\Pi(P_i)$ denote the view of participant P_i during the execution of protocol Π. More precisely, P_i's view is formed by its input and internal random coin tosses r_i, as well as messages $m_1, \ldots, m_k$ passed between the parties during protocol execution: $\mathrm{VIEW}_\Pi(P_i) = (\mathsf{in}_i, r_i, m_1, \ldots, m_k)$. Let $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\}$ denote a subset of the participants for $t < N$, $\mathrm{VIEW}_\Pi(I)$ denote the combined view of participants in I during the execution of protocol Π (i.e., the union of the views of the participants in I), and $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_N)$ denote the projection of $f(\mathsf{in}_1, \ldots, \mathsf{in}_N)$ on the coordinates in I (i.e., $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_N)$ consists of the i_1th, ..., i_tth element that $f(\mathsf{in}_1, \ldots, \mathsf{in}_N)$ outputs). We say that protocol Π is t-private in the presence of semi-honest adversaries if for each coalition of size at most t there exists a probabilistic polynomial time (PPT) simulator S_I such that $\{S_I(\mathsf{in}_I, f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)), f(\mathsf{in}_1, \ldots, \mathsf{in}_n)\} \equiv \{\mathrm{VIEW}_\Pi(I), (\mathsf{out}_1, \ldots, \mathsf{out}_n)\}$, where $\mathsf{in}_I = \bigcup_{P_i \in I}\{\mathsf{in}_i\}$ and $\equiv$ denotes computational or statistical indistinguishability.

The focus of this work is on precise (privacy-preserving) floating-point summation, and this operation is typically a part of a larger computation. For that reason, the inputs into the summation would be the result of other computations on private data. Therefore, we assume that the inputs into the summation are not known by the computational parties and are instead entered into the computation in a privacy-preserving form. Similarly, the output of the summation can be used for further computation and is not disclosed to the parties. In other words, we are developing a building block that can be used in other computations, where the computational parties are given privacy-preserving representation of the inputs, jointly produce a privacy-preserving representation of the output, and must not learn any information about the values they handle. This permits our solution to be used in any higher-level computation and abstracts the setting from the way the inputs are entered into the computation (which can come from the computational parties themselves or external input providers).

In our solution, we heavily rely on the fact that composition of secure building blocks is also secure. As part of this work, we develop several new building blocks to enable the functionality we want to support.

3.2 Secret Sharing

To realize secure computation, we utilize (N, t)-threshold linear secret sharing. Secret sharing offers efficiency due to the information-theoretic nature of the techniques and consequently the ability to operate over a small field or ring. Many of the protocols developed in this work can be realized using any suitable type of secret sharing (e.g,. with or without honest majority and in the semi-honest or malicious settings) and by $[x]$ we denote a secret-shared representation of value x, which is an element of the underlying field or ring. The expected properties are that (i) each of the N computational parties P_i holds its own share such that any combination of t shares reveals no information about x and (ii) a linear combination of secret-shared values can be computed by each party locally on its shares. SPD$\mathbb{Z}_{2^k}$ [19] is one example of a suitable framework.

For performance reasons, many recent publications utilize computation over ring $\mathbb{Z}_{2^k}$ for some $k \geq 1$, which permits the use of native CPU instructions for performing ring operations. This is also the setting that we utilize for our experiments and use to inform certain protocol optimizations. Conventional techniques such as Shamir secret sharing [48] cannot operate over $\mathbb{Z}_{2^k}$ and thus we rely on replicated secret sharing [34] with a small number of parties. Specifically, we use the setting with honest majority, i.e., where $t < N/2$, and are primarily interested in the three-party setting, i.e., $N = 3$. All parties $P_1, \ldots, P_N$ are assumed to be connected by pair-wise secure authenticated channels.

There is a need to secret share both positive and negative integers and the space is used to naturally represent all values as non-negative ring/field elements. In that case, the most significant bit of the representation determines the sign.

For efficiency reasons, portions of the computation proceed on secret shared values set up over a different ring, most commonly $\mathbb{Z}_2$. Thus, we use notation $[x]_\ell$ to denote secret sharing over $\mathbb{Z}_{2^\ell}$ when ℓ differs from the default k.

3.3 Building Blocks

In a linear secret sharing scheme, a linear combination of secret-shared values can be performed locally on the shares without communication. This includes addition, subtraction, and multiplication by a known element. Multiplication of secret-shared values requires communication and the cost varies based on the setting. We use the multiplication protocol from [7] that works with any number of parties in the honest majority setting and communicates only one element in one round in the three-party setting, i.e., when $n = 3$, it matches the cost of three-party protocols such as [5]. Realizing the dot product operation can also often be performed with the communication cost of a single multiplication, regardless of the size of the input vectors.

Our computation additionally relies on the following common building blocks:

- **Equality.** An equality to zero protocol $[b]_1 \leftarrow \mathsf{EQZ}([a])$ takes a private integer input $[a]$ and returns a private bit $[b]$, which is set to 1 if $a = 0$ and is 0 otherwise. Equality of private integers $[x]$ and $[y]$ can be computed by calling the protocol on input $[a] = [x] - [y]$. We use a variant of the protocol from [20] that produces the output bit secret shared over $\mathbb{Z}_2$ (i.e., skips the conversion of the result to the larger ring).
- **Comparisons.** $[b] \leftarrow \mathsf{MSB}([a])$ outputs the most significant bit $[b]$ of its input $[a]$. When working with positive and negative values, MSB computes the sign and is equivalent to the less-than-zero operation. For that reason, the operation can also be used to compare two integers $[x]$ and $[y]$ by supplying their difference as the input into the function. We use the protocol from [7].
- **Bit decomposition.** $[x_{\ell-1}]_1, \ldots, [x_0]_1 \leftarrow \mathsf{BitDec}([x], \ell)$ performs bit decomposition an ℓ-bit input $[x]$ and outputs ℓ secret-shared bits. Our implementation uses the protocol from [20], with a modification that random bit generation is based on edaBits (see below) and the output bits are secret shared over $\mathbb{Z}_2$ by skipping their conversion to $\mathbb{Z}_{2^k}$.
- **Truncation.** Truncation $[y] \leftarrow \mathsf{Trunc}([x], \ell, u)$ takes a secret-shared input $[x]$ at most ℓ bits long and realizes a right shift by u bits. It outputs $y = \lfloor \frac{x}{2^u} \rfloor$. We invoke this function only on non-negative inputs x. Our implementation augments randomized truncation TruncPr from [7] with BitLT implemented using a generic carry propagation mechanism.
- **Prefix AND.** On input $[x_1]_1, \ldots, [x_n]_1$, PrefixAND outputs $[y_1]_1, \ldots, [y_n]_1$, where $y_i = \prod_{j=1}^{i} x_j$. This is the same as $y_i = \bigwedge_{j=1}^{i} x_j$ when x_is are binary. PrefixAND can be

realized as described in [14] using a generic prefix operation procedure (when operating over a ring). As the inputs are bits, for performance reasons this protocol is carried out in $\mathbb{Z}_2$.

- **Prefix OR.** Protocol $[y_1]_1, \ldots, [y_n]_1 \leftarrow \mathsf{PrefixOR}([x_1]_1, \ldots, [x_n]_1)$ produces $y_i = \bigvee_{j-1}^{i} x_j$. This operation can also be implemented using a generic prefix operation mechanism and executed over $\mathbb{Z}_2$.
- **All OR.** $[y_0]_1, \ldots, [y_{2^n-1}]_1 \leftarrow \mathsf{AllOr}([x_{n-1}]_1, \ldots, [x_0]_1)$ takes n bits and produces 2^n bits y_j of the form $\bigvee_{i=0}^{n-1} c_i$, where each c_i is either x_i or its complement $\neg x_i$ and the protocol enumerates all possible combinations. The important property is that only one element at position $x = \prod_{i=0}^{n-1} 2^i x_i$ in the output array will be set to 1, while the remaining elements will be 0. The protocol is described in [9], which we implement over a ring.
- **Random bit generation.** Generation of random bits is a lower-level component of many common building blocks including comparisons, bit decomposition, etc. In this work, we use edaBit from [25] for this purpose. The protocol $[r], [r_{n-1}]_1, \ldots, [r_0]_1 \leftarrow \mathsf{edaBit}(n)$ produces random bits $[r_i]$ shared in $\mathbb{Z}_2$ and the integer they represent $r = \prod_{i=0}^{n-1} 2^i r_i$ in $\mathbb{Z}_{2^k}$.
- **Share reconstruction.** Another lower-level protocol on which we rely is $x = \mathsf{Open}([x], \ell)$ for reconstructing a secret-shared value to the computation participants. To achieve security guarantees, we use a variant that reconstructs $x \in \mathbb{Z}_{2^\ell}$ from $[x]$ where $\ell \leq k$. This is achieved by reducing each share modulo 2^ℓ prior to the reconstruction to guarantee that no information beyond the ℓ bits is exchanged during the reconstruction.
- **Ring conversion.** $[x]_{k'} \leftarrow \mathsf{Convert}([x]_k, k, k')$ starts with x secret-shared over $\mathbb{Z}_{2^k}$ and produces shares of the same value secret-shared over $\mathbb{Z}_{2^{k'}}$, where $k' > k$, i.e., the target ring is larger. We use the Convert protocol from [7].

We also develop several other building blocks as described in Section 4. Note that many of these building blocks can be implemented using different variants, where the mechanism for random bit generation plays a particular role. Using the edaBit approach as described above lowers communication cost of protocols compared to generating each random bit separately with shares in $\mathbb{Z}_{2^k}$, but incurs a higher number of communication rounds. We make design choices in favor of lowering communication, but the alternative is attractive when summing a small number of inputs or when the latency between the computational nodes is high.

Notation $\leftarrow$ is used for functionalities that draw randomness (to produce randomized output or to compute a deterministic functionality that internally uses randomization) and notation = is used for deterministic computation.

4 SECURE LARGE-PRECISION CONSTRUCTION

We are now ready to proceed with our solution for secure and accurate floating-point number summation based on the superaccumulator structure of Section 2.2. As before, a floating-point number x_i is represented as a tuple $\langle b_i, v_i, p_i \rangle$. Our solution takes a sequence of n secret-shared floating-point inputs $\langle [b_i], [v_i], [p_i] \rangle$ and produces a secret-shared floating-point sum. At high level, it proceeds by first converting the inputs into superaccumulators, then computing the sum of the superaccumulators, regularizing the result, and converting the resulting superaccumulator to a floating-point number. The protocol, denoted as FLSum, is given in Algorithm 2 (superaccumulator summation and regularization are combined into SASum). Data representation parameters e, m, and w are fixed throughout the computation (as given in Equation 1) and are implicit inputs.

Algorithm 2 $[s] \leftarrow \mathsf{FLSum}(\langle [b_1], [v_1], [p_1] \rangle, \ldots, \langle [b_n], [v_n], [p_n] \rangle)$

```
let $\alpha = \lceil \frac{2^e+m}{w} \rceil$ and $\beta = \lceil \frac{m+1}{w} \rceil + 1$;
for $i = 1, \ldots, n$ in parallel do
    $\langle [y_{i,\alpha}], \ldots, [y_{i,1}] \rangle \leftarrow \mathsf{FL2SA}([b_i], [v_i], [p_i], \alpha, \beta)$;
end for
$\langle [y_\alpha], \ldots, [y_1] \rangle \leftarrow \mathsf{SASum}(\langle [y_{1,\alpha}], \ldots, [y_{1,1}] \rangle, \ldots, \langle [y_{n,\alpha}], \ldots, [y_{n,1}] \rangle)$;
$\langle [b], [v], [p] \rangle \leftarrow \mathsf{SA2FL}([y_\alpha], \ldots, [y_1])$;
return $\langle [b], [v], [p] \rangle$;
```

When constructing a privacy-preserving solution, the computation that we perform must be data-independent or data-oblivious, as not to disclose any information about the underlying values. In the context of working with the superaccumulator representation, we need to be accessing all superaccumulator slots in the same way regardless of where the relevant data might be located. In particular, when converting a floating-point value to a superaccumulator, at most β slots will contain non-zero values, but their location cannot be disclosed. Similarly, when converting a regularized superaccumulator corresponding to the sum to its floating-point representation, only most significant non-zero slots are of relevance, but we need to hide their position within the superaccumulator.

It is important to note that, unless specified otherwise, the computation is performed over $2w$-bit shares (or ring $\mathbb{Z}_{2^{2w}}$ in our implementation) to facilitate superaccumulator operations. We denote the default element bitlength by k. This default bitlength is sufficient to represent all values with a single exception: the bitlength m mantissa v in the floating-point representation can often exceed the value of $2w$. For that reason, we represent mantissa v as as a sequence of $\lceil \frac{m+1}{w} \rceil$, or $\beta - 1$, secret-shared blocks storing w bits of v per block. For clarity of exposition, each v_i is written as a single shared value in FLSum, while in the more detailed protocols that follow we make this representation explicit.

For most protocols in this paper, including FLSum in Algorithm 2, security follows as a straightforward composition of the building blocks assuming that the sub-protocols are themselves secure. Then using a standard definition of security that requires a simulator without access to private data to produce corrupt parties' view indistinguishable from the protocol's real execution, we can invoke the simulators corresponding to the sub-protocols and obtain security of the overall construction. Thus, in the remainder of this work we discuss security of a specific protocol only when demonstrating its security involves going beyond a simple composition of its sub-protocols. In addition, for some protocols it is important to ensure that they are data-oblivious (i.e., data-independent) such

Algorithm 3 $\langle[y_\alpha],\ldots,[y_1]\rangle \leftarrow \mathsf{FL2SA}([b],\langle[v_{\beta-1}],\ldots,[v_1]\rangle, [p],\alpha,\beta)$

```
1:  $[p^{high}] \leftarrow \mathsf{Trunc}([p], e, \log w)$;
2:  $[p^{low}] = [p] - [p^{high}] \cdot w$;
3:  $[z]_1 \leftarrow \mathsf{EQZ}([p])$;
4:  $[v_{\beta-1}] = [v_{\beta-1}] + 2^{m-w(\beta-2)} \cdot \mathsf{B2A}(1 - [z]_1)$;
5:  $\langle[v_\beta],\ldots,[v_1]\rangle \leftarrow \mathsf{Shift}(\langle[v_{\beta-1}],\ldots,[v_1]\rangle, [p^{low}], w)$;
6:  **for** $i = 1,\ldots,\beta$ in parallel **do**
7:      $[v_i] \leftarrow ([1] - 2 \cdot [b]) \cdot [v_i]$;
8:  **end for**
9:  $\langle[d_\alpha],\ldots,[d_1]\rangle \leftarrow \mathsf{B2U}([p^{high}] + 1, \alpha)$;
10: **for** $i = 1,\ldots,\alpha$ in parallel **do**
11:     **if** $i < \beta$ **then**
12:         $[y_i] \leftarrow \sum_{j=0}^{i}[d_{i-j}] \cdot [v_j]$;
13:     **else if** $i \le \alpha - \beta + 1$ **then**
14:         $[y_i] \leftarrow \sum_{j=0}^{\beta-1}[d_{i-j}] \cdot [v_j]$;
15:     **else**
16:         $[y_i] \leftarrow \sum_{j=0}^{\alpha-1-i}[d_{i-\beta+1+j}] \cdot [v_{\beta-1-j}]$;
17:     **end if**
18: **end for**
19: **return** $\langle[y_\alpha],\ldots,[y_1]\rangle$;
```

that the executed instructions and accessed memory locations are independent of private inputs. Data obliviousness is necessary for achieving security because we need the ability to simulate corrupt parties' view without access to private data.

4.1 Floating-Point to Superaccumulator Conversion

The first component is to convert floating-point inputs to their superaccumulator representation. Because this operation is rather complex and needs to be performed for each input, it dominates the cost of the overall summation and thus it is important to optimize the corresponding computation. The conversion procedure takes a floating point value $([b], \langle[v_{\beta-1}],\ldots,[v_1]\rangle, [p])$ representing normalized $x = (-1)^b \cdot (1 + 2^{-m}v) \cdot 2^{p-2^{e-1}-1}$ and needs to produce a regularized superaccumulator as a vector of α $2w$-bit integers, where $\alpha = \lceil \frac{2^e+m}{w} \rceil$.

4.1.1 The Overall Construction. To perform the conversion, the computation needs to determine the position within the superaccumulator where the mantissa is to be written based on exponent $[p]$, represent the mantissa as β superaccumulator blocks, and write the blocks in the right locations without disclosing what locations within the superaccumulator those are. The protocol details are given as protocol FL2SA (Algorithm 3), which we consequently explain.

Recall that the superaccumulator's step is 2^w. This means that $e - \log w$ most significant bits of the exponent $[p]$ represent the index of the first non-zero slot in the accumulator. The $\log w$ least significant bits of the exponent are used to shift the mantissa so that it is aligned with the block representation of the superaccumulator. Thus, in the beginning of FL2SA we divide the exponent $[p]$ into two parts: the most significant $e - \log w$ are denoted by p^{high} and the remaining $\log w$ bits are denoted by p^{low} (lines 1–2).

The next task is to use the mantissa (represented as $\beta - 1$ blocks) and $[p^{low}]$ to generate β superaccumulator blocks. First, recall that normalized floating-point representation assumes that the most significant bit of the mantissa is 1 and is implicit in the floating-point representation. Thus, we need to prepend 1 as the $(m+1)$st bit of v. In FL2SA we do this conditionally only when the exponent is nonzero (lines 3–4) because when $p = 0$, normalization might not be possible (e.g., if the floating-point value represents a zero). Second, we need to shift the updated mantissa blocks by a private $\log w$-bit value p^{low} to be aligned with the boundaries of superaccumulator blocks and update each value to be w bits by carrying the overflow into the next block.

To perform re-partitioning, we considered solutions based on bit decomposition and truncation for re-partitioning the blocks, and the second approach was determined to be faster. Our final solution – a protocol called Shift that takes the original mantissa blocks – left shifts the values by private $[p^{low}]$ positions, where w is the upper bound on the amount of shift, and re-aligns the blocks to contain w bits each using truncation. The details of the Shift protocol are deferred to the next sub-section. After producing the superaccumulator blocks (line 5), we update the sign of each block using bit $[b]$ (lines 6–8). The desired superaccumulator representation is depicted in Figure 1, where the produced superaccumulator blocks are intended to be written in positions $p^{high} + 1$ through $p^{high} + \beta$.

The last task is to write the generated β superaccumulator blocks $[v_i]$s into the right positions of our α-block superaccumulator, as specified by the value of $[p^{high}]$. Because the computation must be data-oblivious, the location of writing cannot be revealed and the access pattern must be the same for any value of p^{high}. To accomplish the task, we considered two possible solutions: (i) turning the value of p^{high} into a bit array of size α with the p^{high} value set to 1 and all others set to 0 and using the bit array to create superaccumulator blocks and (ii) creating a bit array with a single 1 in the first location and rotating the bit array by a private amount p^{high}. The first approach was determined to be faster and we describe it next.

The conversion of $[p^{high}]+1$, the value of which ranges between 1 and α, to a bit array of private bits with the $(p^{high}+1)$th bit set to 1 can be viewed as binary to unary conversion, denoted by B2U. Prior work considered this building block, and specifically in the context of secure floating-point computation [3], but prior implementations were over a field. Because computation over a ring of the form $\mathbb{Z}_{2^k}$ can be substantially faster, we design a new protocol suitable over a ring using recent results, as described later in this section. After the binary-to-unary conversion of $p^{high} + 1$ (line 9 of FL2SA), each slot of the superaccumulator $[y_i]$ is computed as the dot product of the previously computed data blocks $[v_i]$s and at most β bits $[d_j]$s (lines 10–18) because the data blocks need to be written at positions $p^{high} - \beta + 1$ through p^{high}. In particular, for the middle superaccumulator blocks, there are β bits and data blocks to consider when creating each superaccumulator block $[y_i]$, while the boundary blocks would iterate over fewer options. For example, the block $[y_1]$ will be updated to $[v_1]$ only if $[d_1] = [1]$, while block $[y_2]$ will be updated to $[v_1]$ or $[v_2]$ in the case of

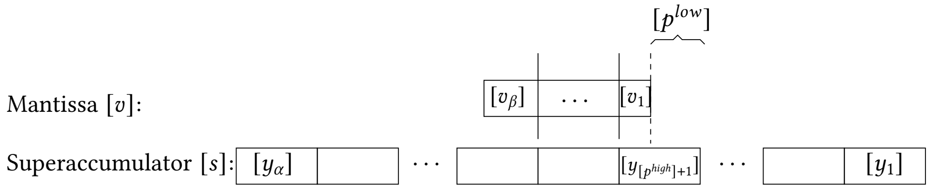

Figure 1: Illustration of floating-point to superaccumulator conversion.

Algorithm 4 $\langle[v_\beta],\ldots,[v_1]\rangle \leftarrow \mathsf{Shift}(\langle[v_{\beta-1}],\ldots,[v_1]\rangle,[p],w)$

```
1:  let $\gamma = \log w$;
2:  $\langle[p_\gamma]_1,\ldots,[p_1]_1\rangle \leftarrow \mathsf{BitDec}([p],\gamma)$;
3:  for $j = 1,\ldots,\gamma$ in parallel do
4:      $[p_j] \leftarrow \mathsf{B2A}([p_j]_1)$;
5:  end for
6:  $[s] \leftarrow \prod_{j=1}^{\gamma}(2^{2^{j-1}}[p_j] + 1 - [p_j])$;
7:  for $i = 1,\ldots,\beta-1$ in parallel do
8:      $[u_i] \leftarrow [v_i][s]$;
9:      $[d_i] \leftarrow \mathsf{Trunc}([u_i],2w,w)$;
10: end for
11: for $i = 2,\ldots,\beta-1$ in parallel do
12:     $[v_i] = [u_i] - 2^w[d_i] + [d_{i-1}]$;
13: end for
14: $[v_1] = [u_1] - 2^w[d_1]$;
15: $[v_\beta] = [d_{\beta-1}]$;
16: return $\langle[v_\beta],\ldots,[v_1]\rangle$;
```

Algorithm 5 $\langle[b_1],\ldots,[b_\ell]\rangle \leftarrow \mathsf{B2U}([a],\ell)$

```
1: $q = \lceil\log\ell\rceil$;
2: $[r],[r_{q-1}]_1,\ldots,[r_0]_1 \leftarrow \mathsf{edaBit}(q)$;
3: $\langle[d_0]_1,\ldots,[d_{2^q-1}]_1\rangle \leftarrow \mathsf{AllOr}([r_{q-1}]_1,\ldots,[r_0]_1)$;
4: $c = \mathsf{Open}([a] - 1 + [r], q)$;
5: for $i = 0,\ldots,\ell-1$ in parallel do
6:     $[b_{i+1}] \leftarrow \mathsf{B2A}(1 - [d_{(c-i) \bmod 2^q}]_1)$;
7: end for
8: return $\langle[b_1],\ldots,[b_\ell]\rangle$;
```

$[d_1] = [1]$ or $[d_2] = [1]$, respectively. All superaccumulator blocks are updated in parallel with communication cost equivalent to that of α multiplications.

4.1.2 New Building Blocks. What remains is to describe our Shift and B2U protocols. The Shift protocol takes an integer value (mantissa in the context of this work) stored in $\beta-1$ blocks $[v_{\beta-1}],\ldots,[v_1]$, shifts the value left by a private amount specified by the second argument $[p]$, where the value of p ranges between 0 and w specified by the third argument, and outputs β new blocks $[v_\beta],\ldots,[v_1]$. It is implicit in the interface specification that each original block representation has (at least) w unused bits, so that the content of each block can be shifted by up to w positions without losing information. In particular, we assume that each block has w bits occupied, so that after the shift the intermediate result can grow to $2w$ bits before being reorganized to occupy w bits per block.

The computation, given in Algorithm 4, starts by bit-decomposing the private amount of shift $[p]$ and converting the resulting bits to ring elements (lines 1–5). The content of each block $[v_i]$ is shifted left (as multiplication by a power of 2) by the appropriate number of positions depending on the value of each bit of the amount of shift: when bit $[p_j]$ is 0, the value is multiplied by 1; otherwise it is multiplied by a power of 2 that depends on the index j (lines 6–8). We then truncate each shifted block (line 9) to split the value into the least significant w bits that the block will retain and the most significant w bits which will are the carry for the next block. Each block is consequently updated by taking the carry from the prior block and keeping its w least significant bits (lines 11–15). Because we shift all blocks in the same way, this operation corresponds to a shift with block re-aligning on the boundary of w bits per block.

Our ring-based solution for binary-to-unary conversion B2U takes a private integer $[a]$ and public range, where $0 < a \leq \ell$, and produces a bit array $\langle[b_1],\ldots,[b_\ell]\rangle$ with the ath bit set to 1 and all other bits set to 0. Our goal is to have a variant suitable for computation over ring $\mathbb{Z}_{2^k}$ using most efficient currently available tools. Our solution, shown as Algorithm 5, is based on ideas used for retrieving an element of an array at a private index in [9].

The high-level idea consists of generating $\lceil\log\ell\rceil$ random bits $[r_i]$ that collectively represent a random $\lceil\log\ell\rceil$-bit integer $[r]$, generating $\lceil\log\ell\rceil$-ary ORs of $[r]-i$ for all $\log\ell$-bit i and flipping the resulting bits. This creates a bit array with all values set to 0 except the element at private location $[r]$ set to 1. The ORs are computed simultaneously for all values using protocol AllOr. Consequently, the algorithm opens the value of $c = r + a$ (modulo $2^{\lceil\log\ell\rceil}$) and uses the disclosed value to position the only 1 bit of the array in location a (i.e., the bit will be set at position i for which $c - i = r + a - i = r$).

Note that the protocol explicitly calls edaBit for random bit generation (and inherits its properties) and there are alternatives. We enhance performance by carrying out the most time-consuming portion of the computation, namely AllOr, over a small ring $\mathbb{Z}_2$ because the computation uses Boolean values. This means that after producing $2^{\lceil\log\ell\rceil}$ bits through a sequence of calls to edaBit, AllOr, and Open and array rotation, we need to convert their shares from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^k}$, which we do using binary-to-arithmetic share conversion B2A (line 6). In addition, reconstruction of $c = r + a$ on line 4 needs to be performed using q-bit shares to enforce modulo reduction and prevent information leakage, where share truncation prior to the reconstruction is performed by Open itself using the modulus specified as the second argument.

As far as security goes, we note that besides composing sub-protocols the protocol also reconstructs a value which is a function of private input $[a]$ on line 8. Security is still achieved because $[r]$ is a private value uniformly distributed in $\mathbb{Z}_{2^q}$. Thus, the value of $[a]$ is perfectly protected and the opened element of $\mathbb{Z}_{2^q}$ is also

Protocol	Communication	Rounds
[20]	$6(k+2)$	2
[4]	$6k$	2
[40]	$6k$	1
Ours	$3k$	2

Table 1: Comparison of three-party B2A **protocols in the honest majority setting with target ring $\mathbb{Z}_{2^k}$. Total protocol communication is reported in bits.**

uniformly distributed over the entire range. This means that the view is easily simulatable by choosing a random element of $\mathbb{Z}_{2^q}$ as the output of Open and getting the parties to reconstruct that value.

The last component that we would like to discuss is the B2A protocol. Solutions for converting a bit b secret shared over $\mathbb{Z}_2$ to the same value secret shared over larger ring $\mathbb{Z}_{2^k}$, $[b]_k \leftarrow \mathsf{B2A}([b]_1, k)$, appear in the literature. Conventional solutions that use square root computation to generate a random bit (e.g., [20]) temporarily increase the ring to be $\mathbb{Z}_{2^{k+2}}$ for computing intermediate results. In the context of this work, this effectively doubles the size of the ring elements during the computation when we use a ring $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$. When the number of participants is not large, an alternative is to cast each local share in $\mathbb{Z}_2$ as a share in $\mathbb{Z}_{2^k}$ and have the parties compute XOR of those values over $\mathbb{Z}_{2^k}$. This approach is used in Araki et al. [4] in the three-party setting with honest majority based on replicated secret sharing (RSS) that costs two consecutive multiplications. The approach of Mohassel and Rindal [40] would also require the same communication in two rounds, but the use of the three-party OT procedure in that work reduces the number of rounds to one.

In this work, we design a new solution in the three-party setting using RSS that does not increase the ring size and lowers the cost of prior protocols as illustrated in Table 1. Unlike many protocols in this work that can be adapted to different types of underlying arithmetic and the number of computational parties, this is the only protocol that specifically uses RSS with $N = 3$ and threshold $t = 1$.

With RSS when $N = 3$, there are three shares representing any secret-shared $[x]$ which we denote as $[x]^{(1)}$, $[x]^{(2)}$, and $[x]^{(3)}$. Each computational party P_i holds two shares with indices different from i. For example, P_2 holds shares $[x]^{(1)}$ and $[x]^{(3)}$. We use notation $[x]_k^{(i)}$ to denote a share in ring $\mathbb{Z}_{2^k}$. In addition, each participant with access to shares indexed by i holds a (sufficiently long) key key_i used as the seed to a pseudorandom generation. For clarity, we refer to a PRG keyed by key_i as G_i. A call to $\mathsf{G}_i.\mathsf{next}$ produces a pseudorandom ring element.

The input to B2A is a bit secret-shared over $\mathbb{Z}_2$ and we need to convert the bit to the shares over $\mathbb{Z}_{2^k}$ as specified by the second argument. The protocol is given as Algorithm 6. The high-level idea behind the solution is that $x = [x]_1^{(1)} \oplus [x]_1^{(2)} \oplus [x]_1^{(3)}$ and we use the knowledge of the input shares by the parties to evaluate the two XOR operations in the target ring. In particular, we can conceptualize the bit shares $[x]_1^{(i)}$ as secret-shared values over $\mathbb{Z}_{2^k}$ represented as $[a]_k = \langle [x]_1^{(1)}, 0, 0\rangle$, $[b]_k = \langle 0, [x]_1^{(2)}, 0\rangle$, and $[c]_k = \langle 0, 0, [x]_1^{(3)}\rangle$. If we securely evaluate $[a]_k \oplus [b]_k \oplus [c]_k$, we will obtain secret-shared $[x]_k$ in the desired ring, which could be generically accomplished by two sequential multiplications (i.e., $[a] \oplus [b] = [a] + [b] - 2[a] \cdot [b]$). This is also the logic used in [4].

Algorithm 6 $[x]_k \leftarrow \mathsf{B2A}([x]_1, k)$

```
Setup: Party $P_i$ holds shares and has access to PRGs $\mathsf{G_j}$ with indices $j \neq i$.
1: set $[a]_k = \langle [x]_1^{(1)}, 0, 0\rangle$, $[b]_k = \langle 0, [x]_1^{(2)}, 0\rangle$, and $[c]_k = \langle 0, 0, [x]_1^{(3)}\rangle$;
2: evaluate $[s]_k = [a]_k \cdot [b]_k$ as follows:
   (a) $P_3$ computes $[s]_k^{(2)} = \mathsf{G}_2.\mathsf{next}$, $[s]_k^{(1)} = [a]_k^{(1)} \cdot [b]_k^{(2)} - [s]_k^{(2)}$ (in $\mathbb{Z}_{2^k}$), and sends $[s]_k^{(1)}$ to $P_2$;
   (b) $P_2$ sets $[s]_k^{(1)}$ to the received value and $[s]_k^{(3)} = 0$;
   (c) $P_1$ computes $[s]_k^{(2)} = \mathsf{G}_2.\mathsf{next}$ and sets $[s]_k^{(3)} = 0$;
3: $[s]_k = [a]_k + [b]_k - 2[s]_k$;
4: evaluate $[u]_k = [s]_k \cdot [c]_k$ as follows:
   (a) $P_2$ computes $[u]_k^{(1)} = \mathsf{G}_1.\mathsf{next}$, $u' = [s]_k^{(1)} \cdot [c]_k^{(3)} - [u]_k^{(1)}$ (in $\mathbb{Z}_{2^k}$), sends $u'$ to $P_1$, and computes $[u]_k^{(3)} = u' + \mathsf{G}_3.\mathsf{next}$ (in $\mathbb{Z}_{2^k}$).
   (b) $P_1$ receives $u'$, computes $u'' = \mathsf{G}_3.\mathsf{next}$, $[u]_k^{(2)} = [s]_k^{(2)} \cdot [c]_1^{(3)} - u''$, and $[u]_k^{(3)} = u' + u''$ (all computation is in $\mathbb{Z}_{2^k}$), and sends $[u]_k^{(2)}$ to $P_3$;
   (c) $P_3$ sets $[u]_k^{(2)}$ to the received value and $[u]_k^{(1)} = \mathsf{G}_1.\mathsf{next}$.
5: $[x]_k = [s]_k + [c]_k - 2[u]_k$;
6: return $[x]_k$;
```

However, given that our shares of a, b, and c have a special form, the cost of that computation can be reduced. In particular, a typical implementation of the multiplication operation involves multiplying accessible shares locally and re-sharing the products with other parties using fresh randomization to hide patterns. Because in our case some shares are set to 0, their product will be 0 as well, and no re-sharing is needed. For example, when computing $[a]_k \cdot [b]_k$, the only contributing term to the product is the product of $[a]_k^{(1)}$ and $[b]^{(2)}$, which is computable by P_3 in its entirety. As a result of such optimizations, the communication cost of the overall protocol is one ring element per party.

Referring to Algorithm 6, as mentioned above, the product of $[a]$ and $[b]$ (step 2) can be computed locally by P_3, after which the product is re-shared. The re-sharing uses proper k-bit elements to hide information about the product and is split by P_3 in two shares to which it has access, namely $[s]^{(1)}$ and $s^{(2)}$. This is similar to the re-sharing in regular multiplication (see, e.g., [7]) and involves P_3 communicating a single ring element.

After turning the product into XOR (line 3), the parties need to compute the product of $[s]_k$ and $[c]_k$, where $[s]_k$ has two non-empty shares ($[s]_k^{(1)}$ and $[s]_k^{(2)}$) and $[c]_k$ has one non-empty share ($[c]_k^{(3)}$). This involves P_2 computing the product $[s]_k^{(1)} \cdot [c]_k^{(3)}$ and re-sharing by splitting it into two shares and P_1 computing the product $[s]_k^{(2)} \cdot [c]_1^{(3)}$ and also re-sharing it. As described in step 4 of Algorithm 6, P_2's product is split into $[u]_k^{(1)}$ and value u', which becomes a part of $[u]_k^{(3)}$. Similarly, P_1's product is split into $[u]_k^{(2)}$ and value u'', which becomes the second component of $[u]_k^{(3)}$. Both P_1 and P_2 communicate one ring element each to finish re-sharing and let everyone obtains the shares of the product u. The party then finish the computation by turning the product into XOR (line 5). The total communication is equivalent to that of a single multiplication.

We prove the following result:

Claim 1. B2A *protocol in Algorithm 6 is 1-private in the semi-honest model in the three-party setting in the presence of a single computationally-bounded corrupt party assuming* G *is a pseudo-random generator.*

Proof. We prove that our B2A protocol in Algorithm 6 is secure in the presence of a single corrupt party. We consider corruption of party P_1, P_2, and P_3 in turn and build a corresponding simulator for each case.

Party P_1 is corrupt. We first assume that party P_1 is corrupt, and build the corresponding simulator S_1 to simulate its view in the ideal model. The simulator S_1 is constructed as follows:

- In step 4(a), S_1 draws a uniformly random element $u' \leftarrow \mathbb{Z}_{2^k}$ and sends it to party P_1 on behalf of party P_2.
- In step 4(b), S_1 receives $[u]_k^{(2)}$ from P_1 on behalf of P_3.

We next compare the view of P_1 that the simulator S_1 produces with the view of the corrupt party P_1 in the real execution. In the beginning of the protocol, P_1 holds $[b]_k^{(2)} = [x]_1^{(2)}$ and $[c]_k^{(3)} = [x]_1^{(3)}$ and has access to G_2 and G_3. The simulated view consists of P_1 receiving a randomly generated u' in step 4(a), while in the real execution it was computed as $u' = [s]_k^{(1)} \cdot [c]_k^{(1)} - G_1.\mathrm{next}$. Now because P_1 does not have access to G_1, the pseudo-random pad G_1.next information-theoretically protects the value of the product $[s]_k^{(1)} \cdot [c]_k^{(1)}$. Thus, the value of u' in the real execution is pseudo-random. Then because by definition of a pseudo-random generator its output is indistinguishable from a truly random string of the same size to a computationally-bounded adversary, we obtain that the simulated and real views are indistinguishable.

Party P_2 is corrupt. Next, consider the case that party P_2 is corrupt. We construct simulator S_2 as follows:

- In step 2(a), S_2 draws a uniformly random $[s]_k^{(1)} \leftarrow \mathbb{Z}_{2^k}$ and sends it to P_2 on behalf of party P_3.
- In step 4(b), S_2 receives u' from P_2 on behalf of P_1.

At computation initiation time, P_2 holds $[a]_k^{(1)} = [x]_1^{(1)}$ and $[c]_k^{(3)} = [x]_1^{(3)}$ and has access to G_1 and G_3. Similar to the case of corrupt P_1, S_2 only communicates a random value as $[s]_k^{(1)}$ to P_2 in step 2(a). In a real execution, $[s]_k^{(1)}$ is computed as $[a]_k^{(1)} \cdot [b]_k^{(2)} - G_2.\mathrm{next}$, where G_2 is inaccessible to P_2 and thus its output information-theoretically protects the product. Because the PRG's output is computationally indistinguishable from a truly random string to a computationally-bounded adversary, P_2's simulated view is computationally indistinguishable from the view in the real execution.

Party P_3 is corrupt. Finally, we construct simulator S_3 for the case that party P_3 is corrupt:

- In step 2(a), S_3 receives $[s]_k^{(1)}$ from party P_3 on behalf of party P_2.
- In step 4(b), S_3 draws a uniformly random value $[u]_k^{(2)} \leftarrow \mathbb{Z}_{2^k}$ and sends it to P_3 on behalf of P_1.

In the beginning of the computation, P_3 has access to $[a]_k^{(1)} = [x]_1^{(1)}$, $[b]_k^{(2)} = [x]_1^{(2)}$, G_1, and G_2. It then receives a random $[u]_k^{(2)}$ from S_3 in the simulated view, while in the real execution the value is computed as $u' + u''$, where $u'' = G_3.\mathrm{next}$. Due to security of the PRG, its output is pseudo-random and information-theoretically protects u'. We obtain that the value P_3 is indistinguishable from a truly random string to a computationally-bounded P_3. Thus, we obtain that P_3's views in real execution and simulation are computationally indistinguishable.

We conclude that our B2A protocol is secure in the presence of a single semi-honest adversary. □

Algorithm 7 $\langle[y_\alpha], \ldots, [y_1]\rangle \leftarrow \mathsf{SASum}(\langle[y_{1,\alpha}], \ldots, [y_{1,1}]\rangle, \ldots, \langle[y_{n,\alpha}], \ldots, [y_{n,1}]\rangle)$

```
for $i = 1, \ldots, \alpha$ in parallel do
    $[s_i] = \sum_{j=1}^{n} [y_{j,i}]$;
    $[b_i] \leftarrow \mathsf{MSB}([s_i])$;
    $[y_i] \leftarrow [s_i] \cdot (2[b_i] - 1)$;
    $[c_{i+1}] \leftarrow \mathsf{Trunc}([y_i], 2w, w)$;
    $[r_i] = [y_i] - [c_{i+1}] \cdot 2^w$;
    $[y_i] \leftarrow [r_i] \cdot [b_i] + [c_i] \cdot [b_{i-1}]$
end for
return $\langle[y_\alpha], \ldots, [y_1]\rangle$;
```

B2A is an important building blocks of many other protocols including truncation, ring conversion, bit decomposition, etc. Thus, the above efficient three-party B2A impact performance of the computation. For that reason, we analyze performance of building blocks and our protocols in the three-party setting using RSS as given in Table 2. Note that we separate input-independent computation that can be pre-computed and the remaining (input-dependent) computation.

Random bit generation $[r] \leftarrow \mathsf{RandBit}$ (as used, e.g., in MSB) is implemented by using local randomness to generate shares of $[r]_1$ over $\mathbb{Z}_2$ and converting them to the larger ring using B2A. We favor the use of edaBit in sub-protocols in place of conventional RandBit random bit generation. This lowers the amount of communication, but increases the number of rounds.

The cost of AllOr as specified in [9] varies based on the size given as an input. For that reason, in Table 2 we list a range of constants for values α used with single and double precision in this work (the smallest $\alpha = 9$ with single precision and $w = 32$ results in constant 1.5 and the largest $\alpha = 132$ with double precision and $w = 16$ results in constant 1.2).

4.2 Superaccumulator Summation

Once we convert the floating-point inputs into superaccumulators, the next step is to do the summation and regularize the result. This corresponds to the protocol SASum given in Algorithm 7. The summation of superaccumulators is straightforward, where we sum each superaccumulator block as $[s_i] = \sum_{i=1}^{n} [y_{i,j}]$ for $i = 1, \ldots, \alpha$

Protocol	Precomputable		After precomputation	
	Communication	Rounds	Communication	Rounds
Mult	0	0	$3k$	1
Open(ℓ), $\ell \le k$	0	0	3ℓ	1
B2A	0	0	$3k$	2
RandBit	$3k$	2	0	0
edaBit(k)	$3k\log(k) + 7k$	$\log(k) + 2$	0	0
edaBit(ℓ), $\ell < k$	$3\ell\log(\ell) + 5\ell + 5k$	$\log(\ell) + 4$	0	0
PrefixOr(n) (in $\mathbb{Z}_2$)	0	0	$1.5n\log(n)$	$\log(n)$
PrefixAnd(n) (in $\mathbb{Z}_2$)	0	0	$1.5n\log(n)$	$\log(n)$
MSB(k)	$3k\log(k) + 10k$	$\log(k) + 2$	$12k - 12$	$\log(k) + 2$
EQZ(k)	$3k\log(k) + 7k$	$\log(k) + 2$	$6k - 3$	$\log(k) + 1$
Trunc(ℓ, u)	$3k\log(k) + 18k$	$\log(u) + 3$	$3k + 3\ell + 6u - 6$	$\log(u) + 3$
BitDec(ℓ), $\ell < k$	$3\ell\log(\ell) + 5\ell + 5k$	$\log(\ell) + 4$	$3\ell\log(\ell) + 3\ell$	$\log(\ell) + 1$
Convert(k, k')	$3k\log(k) + 7k$	$\log(k) + 2$	$3k'k + 3k\log(k) + 3k$	$\log(k) + 3$
Shift(β, w)	$(\beta - 1)(3k\log(k) + 18k)+$ $3\gamma\log(\gamma) + 5\gamma + 5k$	$\max(\log(\gamma) + 4,$ $\gamma + 3)$	$6(\beta - 1)(2k + w - 1)+$ $3\gamma(k + \log(\gamma) + 1) - 3k$	$\gamma + 2\log(\gamma) + 7$
B2U(α)	$[1.2–1.5]3 \cdot 2^{\delta} + 3\delta\log(\delta) + 5\delta + 5k$	$2\log(\delta) + 4$	$3\alpha k + 3\delta$	3
Normalize(β, w)	$\beta(3k\log(k) + 7k)+$ $6l\log(l) + 17l$	$\max(\log(k) + 2,$ $\log(l) + 2)$	$3k(\beta l + \beta\log(k) + l + \beta + m + 1)+$ $1.5(l - m - 2)\log(l - m - 2)+$ $3l(\log(l) + 6) - 12$	$2\log(l) + \log(k)+$ $\log(l - m - 2) + 10$

Table 2: Performance of protocols in the three-party setting based on RSS using ring $\mathbb{Z}_{2^k}$ (bit-level operations are over $\mathbb{Z}_2$). Protocol parameters affecting performance are listed. Total communication across all parties in bits. $\gamma = \lceil\log(w)\rceil$, $\delta = \lceil\log(\alpha)\rceil$, **and** $l = w\beta$; α, β, w, **and** m **are computation parameters.**

(line 2). The remaining computation regularizes the resulting superaccumulator. We first compute the absolute value of each block y_i (lines 3–4) and then split the result into w most significant bits (carry for the next block $[c_{i+1}]$) and w least significant bits ($[r_i]$) using truncation (lines 5–6). The final block value is assembled from the carry of the prior block and the remaining portion of the current block using their corresponding signs (line 7). The carry into block 1 is 0.

Recall that each superaccumulator block is represented as a $2w$-bit integer and we can add at most $n = 2^{w-2}$ inputs without an overflow. If one needs to sum more than 2^{w-2} inputs, the computation will proceed in layers, where we first sum accumulators in batches of 2^{w-2}, regularize the result and then do another layer of summation and regularization to arrive at the final regularized superaccumulator.

4.3 Superaccumulator to Floating-Point Conversion

What remains to discuss is the conversion of the regularized superaccumulator representing the summation to the floating-point representation. To maintain security, our protocols needs to obliviously select β superaccumulator blocks starting from the first non-zero block without disclosing the location of the selected blocks. In the event that there are fewer than β blocks to extract, the solution will still return β blocks.

The superaccumulator to floating-point conversion protocol SA2FL is given as Algorithm 8 and proceeds as follows. Let ind denote the (private) index of the first non-zero superaccumulator block. We restrict the value we work with to be in the range $\alpha, \ldots, \beta$ to ensure that we can always extract β blocks, i.e., ind $= \beta$ even if the first non-zero block has the index smaller than β. Given a regularized superaccumulator $[y_\alpha], \ldots, [y_1]$, we first test each block with the index between β and α for equality to zero. Once it is determined which blocks are zero, we need to compute the prefix AND of the computed bits (or, equivalently, the prefix OR of their complements) to determine the first non-zero block. Recall that

Algorithm 8 $\langle[b], [v_{\beta-1}], \ldots, [v_1], [p]\rangle \leftarrow$ SA2FL($[y_\alpha], \ldots, [y_1]$)

```
1:  for $i = \beta, \ldots, \alpha$ in parallel do
2:      $[c_i]_1 \leftarrow$ EQZ($[y_i]$);
3:  end for
4:  $\langle[d_\alpha]_1, \ldots, [d_{\beta+1}]_1\rangle \leftarrow$ PrefixAND($[c_\alpha]_1, \ldots, [c_{\beta+1}]_1$);
5:  for $i = \beta, \ldots, \alpha$ in parallel do
6:      $[d_i] \leftarrow$ B2A($[d_i]_1$);
7:  end for
8:  for $i = \beta + 1, \ldots, \alpha - 1$ in parallel do
9:      $[u_i] = [d_{i+1}] - [d_i]$;
10: end for
11: $[u_\alpha] = 1 - [d_\alpha]$;
12: $[u_\beta] = [d_{\beta+1}]$;
13: for $i = 1, \ldots, \beta$ in parallel do
14:     $[v_i] \leftarrow \sum_{j=i}^{\alpha-\beta+i} [u_{j+\beta-1-i}] \cdot [y_j]$;
15: end for
16: $\langle[b], [v_{\beta-1}], \ldots, [v_1], [p]\rangle \leftarrow$ Normalize($[v_\beta], \ldots, [v_1]$);
17: $[p] = [p] + \sum_{i=1}^{\alpha-\beta+1} [u_{i+\beta-1}] \cdot i \cdot w$;
18: return $\langle[b], [v_{\beta-1}], \ldots, [v_1], [p]\rangle$;
```

PrefixAND, on input $[x_1], \ldots, [x_n]$ outputs $[y_1], \ldots, [y_n]$, where $y_i = \prod_{j=1}^{i} x_j$. Also, for performance reasons, we do not convert the resulting bits of equality comparisons to full ring element and instead proceed with prefix computation on bits.

For prefix AND, we start with the highest index and thus the output will be a sequence of 1s followed by 0s starting from the high indices. The first 0 is the value we want to mark differently from others, indicating the first non-zero block. This is accomplished by computing the difference between two adjacent block values (lines 8–12) and we obtain the first non-zero block marked with 1, while all other blocks are as 0. It is important to note that the βth block will be marked even if all of the blocks $\alpha, \ldots, \beta$ are 0, because in that case we still need to retrieve β blocks with the smallest values, i.e., ind is set to β and the actual content of the βth block is irrelevant.

The next step is to extract β blocks starting from the marked block, i.e., using the previously introduced notation, we extract the blocks $[y_{\mathsf{ind}}], \ldots, [y_{\mathsf{ind}-\beta+1}]$ (lines 13–15). We consequently normalize the block using a sub-protocol Normalize that returns a floating-point representation of the blocks, which is consequently updated on line 17 to modify the exponent according to the position of the extracted blocks in the superaccumulator.

The next protocol, Normalize, corresponds to the conversion of β extracted superaccumulator blocks to a normalized floating-point value. As before, each block $[v_i]$ is assumed to contain w bits and we normalize the value by finding the first non-zero bit and creating an m-bit mantissa with the $(m+1)$st bit set to 1 and the remaining bits partitioned among the output blocks $[v_{\beta-1}], \ldots, [v_1]$.

The protocol is given as Algorithm 9 and proceeds as follows. The first portion of the computation is concerned with assembling the input blocks as a single integer and consequently determining the first non-zero bit. A complicating factor is that different blocks can have different signs, which makes it non-trivial to work at the level of individual blocks. Therefore, the first step of the computation is to convert the shares of the input blocks from the ring with $k = 2w$-bit elements to longer $l = w\beta$-bit elements (lines 2–4). The blocks are consequently added together as $[s]_l$ (line 5) and the absolute value of $[s]_l$ is computed as $[v]_l$ (lines 6–7). We next bit-decompose the computed value (line 8) and from this point on the computation can return to shorter k-bit shares, but we additionally optimize the computation to run skip immediate conversion of bits to k-bit shares and run the next step on bit shares as well.

Given the bits of the value we need to normalize, we determine the first non-zero bit and grab the next m bits (as the $(m+1)$st bit is 1 and is implicit). If there are fewer than $m+1$ non-zero bits, the value must correspond to the lowest blocks of the superaccumulator (as otherwise, the $w\beta$ bits are guaranteed to contain $m+1$ non-zero bits) and cannot be represented in the properly normalized form. In that case we store the m least significant bits in the mantissa and the floating-point value's exponent will be 0. Thus, we first call the prefix OR operation on the most significant $\approx l - m$ bits (line 9) and compute the difference between the adjacent bits. As a result, the most significant non-zero bit of v will be set to 1 in $[z_i]$s, with all others set to 0 (lines 13–16). If the first non-zero bit is at position m (when counting from 0) or a lower index, z_m is set to 1 to permit retrieval of m least significant bits (line 17). Then the m bits after the marked bit are extracted (lines 18–20) and are stored in $\beta - 1$ blocks (lines 21–24).

Algorithm 9 $\langle [b], [v_{\beta-1}], \ldots, [v_1], [p] \rangle \leftarrow \mathsf{Normalize}([v_\beta], \ldots, [v_1])$

```
1: let $l = w \cdot \beta$;
2: for $i = 1, \ldots, \beta$ in parallel do
3:     $[v_i]_l \leftarrow \mathsf{Convert}([v], k, l)$;
4: end for
5: $[s]_l = \sum_{i=1}^{\beta} 2^{w(i-1)} [v_i]_l$;
6: $[b]_l \leftarrow 1 - 2 \cdot \mathsf{MSB}([s]_l)$;
7: $[v]_l \leftarrow [b]_l \cdot [s]_l$;
8: $\langle [c_{l-1}]_1, \ldots, [c_0]_1 \rangle \leftarrow \mathsf{BitDec}([v]_l, l)$;
9: $\langle [c_{l-2}]_1, \ldots, [c_{m+1}]_1 \rangle \leftarrow \mathsf{PrefixOR}([c_{l-2}]_1, \ldots, [c_{m+1}]_1)$;
10: for $i = 0, \ldots, l-1$ in parallel do
11:     $[c_i] \leftarrow \mathsf{B2A}([c_i]_1)$;
12: end for
13: $[z_{l-1}] = [c_{l-1}]$;
14: for $i = m, \ldots, l-2$ in parallel do
15:     $[z_i] = [c_i] - [c_{i+1}]$;
16: end for
17: $[z_m] = 1 - [c_{m+1}]$;
18: for $i = 0, \ldots, m-1$ in parallel do
19:     $[u_i] \leftarrow \sum_{j=i}^{l-1-m+i} [z_{i+m-i}] \cdot [c_j]$;
20: end for
21: for $i = 1, \ldots, \beta - 2$ do
22:     $[v_i] = \sum_{j=0}^{w-1} [u_{j+i\cdot w}] \cdot 2^j$;
23: end for
24: $[v_{\beta-1}] = \sum_{i=w(\beta-2)}^{m-1} [u_i] \cdot 2^{i-w(\beta-2)}$;
25: $[z_m] \leftarrow [z_m] \cdot [c_m]$;
26: $[p] = \sum_{i=0}^{l-m-1} i \cdot [z_{j+m}]$
27: return $\langle [b], [v_{\beta-1}], \ldots, [v_1], [p] \rangle$;
```

What remains is to form the exponent based on the position of the first non-zero bit. This time we need to distinguish between normalized $(m+1)$-bit mantissas that start from position m and mantissas with fewer than $m+1$ non-zero bits. For that reason, we update the bit $[z_m]$ (line 25) prior to computing the exponent $[p]$ (line 26).

5 PERFORMANCE EVALUATION

In this section, we evaluate performance of our construction and compare it to the state-of-the-art secure floating-point summation protocols. Our implementation is in C++ using RSS over a ring $\mathbb{Z}_{2^k}$ and is available at [1]. We run all experiments in a three-party setting using machines with a 2.1GHz CPU connected by a 1Gbps link with one-way latency of 0.08ms. All experiments are single threaded and are not optimized for round complexity with respect to pre-processing. Instead, randomness generation is performed inline as specified in the protocols and the actual number of rounds in the implementation is higher than what is possible and what is reported in Table 2. Each experiment was executed at least 100 times, and the average runtime is reported.

To evaluate the impact of our new three-party B2A protocol, in Figure 2 we provide performance comparison of a common square root based solution from [20] and our solution described in Algorithm 6. Because the former requires a slightly larger ring

Prot.	Input size															
	w = 16								w = 32							
	2^4	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^4	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
FL2SA	6.81	9.06	16.9	45.4	136	529	2160	8403	6.02	8.13	17.5	46.4	142	585	2324	9036
SASum	3.22	3.14	3.81	3.68	3.56	4.37	20.5	85.1	3.1	3.17	3.68	3.71	3.89	4.17	18.4	79.2
SA2FL	6.82	6.75	6.74	6.67	6.48	6.74	6.87	6.71	7.83	7.84	7.84	7.86	7.74	7.89	7.91	7.74
Total	16.8	18.9	27.4	55.7	146	540	2187	8495	16.9	19.1	28.7	57.9	154	598	2351	9124

(a) Single floating-point precision.

Prot.	Input size															
	w = 16								w = 32							
	2^4	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^4	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
FL2SA	9.09	14.3	36.2	114	413	1668	6688	24805	9.32	14.3	32.9	106.9	384	1517	6247	23486
SASum	4.43	4.87	4.91	4.93	6.29	10.4	17.4	57.4	4.78	4.87	5.17	5.07	6.39	9.71	14.7	45.0
SA2FL	8.78	8.49	8.41	8.31	8.12	8.22	8.25	8.24	9.31	9.14	9.13	8.97	9.04	8.87	8.71	9.23
Total	22.3	27.7	49.5	127	427	1687	6714	24871	23.4	28.3	47.2	121	399	1536	6271	23540

(b) Double floating-point precision.

Table 3: Performance FLSum in ms.

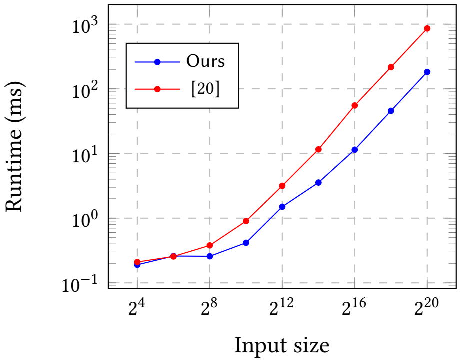

Figure 2: Performance comparison of B2A protocols.

size $\mathbb{Z}_{2^{k+2}}$, we set the computation over $\mathbb{Z}_{2^k} = \mathbb{Z}_{2^{60}}$ and thus portion of the computation for the protocol from [20] are over $\mathbb{Z}_{2^{62}}$. The implications are that both protocols can internally use 64-bit arithmetic and the increase in the ring size does not impact communication in bytes. Therefore, communication and the number of rounds of the protocol from [20] are also the same as those numbers for the protocol from [4]. Had we chosen $k = 32$ or $k = 64$, the gap in performance between our protocol and that from [20] would increase due to the need of the later to increase the communication size and use a longer data type for the computation.

As we see from Figure 2, for smaller input sizes, both solutions exhibit similar performance due to their equivalent round complexity. However, as input size increases beyond 2^6 and communication and computation become dominant factors in overall performance, our solution outperforms [20] by a significant margin. For instance, the performance gap between the two approaches is as large as a factor of four for input size 2^{20}, demonstrating the advantage of our B2A protocol even beyond savings in communication.

Performance of our superaccumulator-based floating-point summation for single and double floating-point precision is provided in Table 3. The performance is additionally visualized in Figures 3 and 4. We see that the bottleneck of the summation for both single and double precision is the conversion FL2SA, particularly when the input size n is large. This is expected because we need to convert all n inputs into the superaccumulator representation. In contrast, superaccumulator to floating-point conversion SA2FL has a constant runtime for all input sizes because we only need to convert a single result and the workload does not change. Although summation SASum has communication complexity independent of n, its local computation linearly depends on the input size, which makes its runtime increase with n.

If we compare the runtimes for different values of w, using $w = 16$ results in lower overall runtime with single precision, while $w = 32$ is superior for double precision. The difference in performance mainly stems from the impact of the choice of w on the performance of FL2SA and its dependence on parameters α and β (which w directly influences).

We also compare performance of our superaccumulator-based solution with floating-point summations from [11, 12, 44]. We execute SecFloat's [44] pairwise addition in a tree-like manner to realize floating-point summation and measure the performance on our setup. Note that SecFloat is for the two-party setting (dishonest majority) and was implemented only for single precision. We also include published runtimes of the best performing solution, SumFL2, from [12] as the implementation has not been released. The experiments in [12] were run using three 3.6GHz machines connected via a 1Gbps LAN, where the round-trip time (RTT) measured via ping was reported to be 0.35ms (our RTT measured via ping averaged at 0.25ms). We also calculate the communication cost of SumFL2 using the specified formula.[1]. The results are given

[1] In [12], communication measured from the implementation differed from communication derived from the analysis and the implementation's communication is 9.3% lower of the analytical cost. Because the measurement included only one data point with 10 operands, we report results computed according to the formula

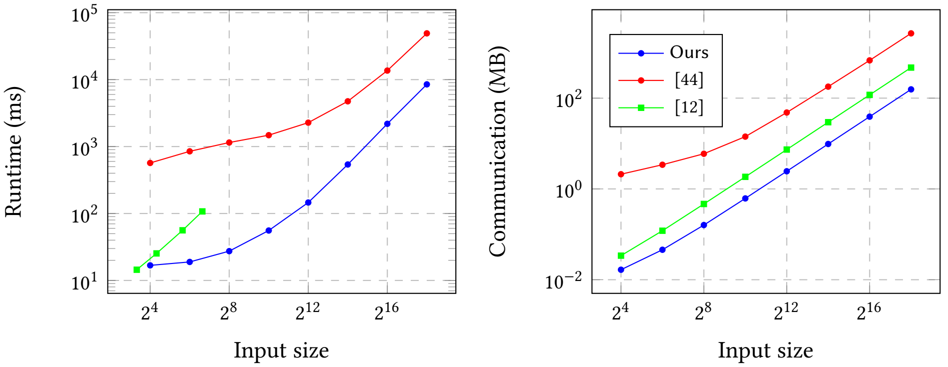

Figure 3: Performance comparison with related work for single precision. [9]'s runtime uses different hardware.

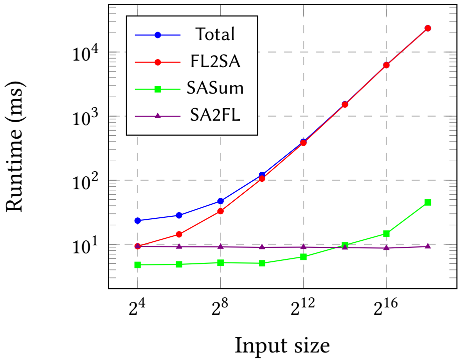

Figure 4: Performance of double precision protocol with w = 32.

Prot.	Input size							
	Single				Double			
	10	20	50	100	10	20	50	100
Ours	16.5	17.4	18.6	21.2	22.3	24.1	27.3	30.5
[12]	14.5	25.3	56.1	107.5	26.5	43.8	95.4	158

Table 4: Runtime comparison with SumFL2 from [12] in ms.

in Figure 3, where our single-precision solution uses $w = 16$.

As shown in the figure, our protocol has better runtime and communication costs than the other two solutions. Although [44] states that their implementation is not optimized for batch sizes smaller than 2^{10}, our protocol is still 5 times faster and uses 17 times less communication than [44] with 2^{18} inputs. For input sizes larger than 2^{14}, both solutions demonstrate the same trend. We expect our advantages would be larger in the WAN setting where bandwidth is limited and communication is the bottleneck.

Compared to [12], our best performing configuration has a better runtime despite running on slower machines, as additionally shown in Table 4. In [12], performance is reported with at most 100 inputs. When $n = 100$, our solution demonstrates the largest improvement, being 5 times faster than SumFL2 from [12] for both single and double precisions. We expect the improvement to be even larger as the number of inputs increases. Furthermore, we note that our solution enjoys higher precision, as the goal of this work was to provide better precision than what is achievable using conventional floating-point addition. Lastly, while [13] discussed additional optimizations to floating-point polynomial evaluation, it is difficult to extract times that would correspond to the summation.

6 CONCLUSIONS

The goal of this work is to develop secure protocols for accurate summation of many floating-point values that avoid round-off errors of conventional floating-point addition. Our solution uses the notion of a superaccumulator and the computation proceeds by converting floating-point inputs into superaccumulator representation, performing exact summation, and converting the computed result back to a floating-point value. Despite providing higher accuracy, we demonstrate that our solution outperforms state-of-the-art secure floating-point summation.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants 2213057 and 2212129. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding sources. The authors would also like to thank anonymous reviewers and Alessandro Baccarini for help with running the experiments.

REFERENCES

[1] Secure floating-point summation implementation. https://github.com/chennyc/floating_point_summation.

[2] M. Aliasgari, M. Blanton, and F. Bayatbabolghani. Secure computation of hidden markov models and secure floating-point arithmetic in the malicious model. *International Journal of Information Security*, 16(6):577–601, 2017.

[3] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. Secure computation on floating point numbers. In *Network and Distributed System Security Symposium*, 2013.

[4] T. Araki, A. Barak, J. Furukawa, M. Keller, K. Ohara, and H. Tsuchida. How to choose suitable secure multiparty computation using generalized SPDZ. In *Poster at ACM Conference on Computer and Communications Security (CCS)*, pages 2198–2200, 2018.

[5] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM Conference on Computer and Communications Security (CCS)*, pages 805–817, 2016.

[6] D. Archer, S. Atapoor, and N. Smart. The cost of IEEE arithmetic in secure computation. In *International Conference on Cryptology and Information Security in Latin America*, pages 431–452, 2021.

[7] A. Baccarini, M. Blanton, and C. Yuan. Multi-party replicated secret sharing over a ring with applications to privacy-preserving machine learning. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2023(1):608–626, 2023.

[8] F. Bayatbabolghani, M. Blanton, M. Aliasgari, and M. Goodrich. Secure fingerprint alignment and matching protocols. arXiv Report 1702.03379, 2017.

[9] M. Blanton, A. Kang, and C. Yuan. Improved building blocks for secure multi-party computation based on secret sharing with honest majority. In *Applied Cryptography and Network Security (ACNS)*, pages 377–397, 2020.

[10] C. Burnikel et al. Exact Geometric Computation in LEDA. In *Symposium on Computational Geometry (SoCG)*, pages 418–419, 1995.

[11] O. Catrina. Optimizing secure floating-point arithmetic: Sums, dot products, and polynomials. In *Proceedings of the Romanian Academy*, volume 21, pages 21–28, 2020.

[12] O. Catrina. Performance analysis of secure floating-point sums and dot products. In *International Conference on Communications (COMM)*, pages 465–470, 2020.

[13] O. Catrina. Complexity and performance of secure floating-point polynomial evaluation protocols. In *European Symposium on Research in Computer Security (ESORICS)*, pages 352–369, 2021.

[14] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In *SCN*, pages 182–199, 2010.

[15] O. Catrina and S. De Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In *European Symposium on Research in Computer Security (ESORICS)*, pages 134–150, 2010.

[16] O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography and Data Security (FC)*, pages 35–50, 2010.

[17] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. A Reproducible Accurate Summation Algorithm for High-Performance Computing. In *SIAM EX14 Workshop*, 2014.

[18] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. Full-speed deterministic bit-accurate parallel floating-point summation on multi- and many-core architectures. *HAL-CCSD, Tech. Rep. hal-00949355*, 2014.

[19] R. Cramer, I. Damgard, D. Escudero, P. Scholl, and C. Xing. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *Advances in Cryptology – CRYPTO*, pages 769–798, 2018.

[20] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1102–1120, 2019.

[21] J. Demmel and Y. Hida. Accurate and efficient floating point summation. *SIAM J. on Scientific Computing*, 25(4):1214–1248, 2004.

[22] J. Demmel and Y. Hida. Fast and accurate floating point summation with application to computational geometry. *Numerical Algorithms*, 37(1-4):101–112, 2004.

[23] J. Demmel and H. D. Nguyen. Parallel reproducible summation. *IEEE TC*, 64(7):2060–2070, July 2015.

[24] V. Dimitrov, L. Kerik, T. Krips, J. Randmets, and J. Willemson. Alternative implementations of secure real numbers. In *ACM Conference on Computer and Communications Security (CCS)*, pages 553–564, 2016.

[25] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *Advances in Cryptology – CRYPTO*, pages 823–852, 2020.

[26] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh. Numerical behavior of NVIDIA tensor cores. *PeerJ Computer Science*, 7:e330, 2021.

[27] L. Fousse et al. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 2007.

[28] M. Franz and S. Katzenbeisser. Processing encrypted floating point signals. In *ACM Multimedia Workshop on Multimedia and Security*, pages 103–108, 2011.

[29] GMPLib. GMP: the GNU multiple precision arithmetic library. https://gmplib.org/. Accessed 2015-12-16.

[30] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, pages 5–48, Mar. 1991.

[31] G. Hanrot, V. Lefévre, P. Pélissier, P. Théveny, and P. Zimmermann. The GNU MPFR library. http://www.mpfr.org/. Accessed 2015-12-16.

[32] M. Hu, J. P. Strachan, Z. Li, R. Stanley, and Williams. Dot-product engine as computing memory to accelerate machine learning algorithms. In *17th Int. Symp. on Quality Electronic Design (ISQED)*, pages 374–379, 2016.

[33] M. Hummel and K. v. Kooten. Leveraging NVIDIA Omniverse for in situ visualization. In *Int. Conf. on High Performance Computing*, pages 634–642, 2019.

[34] M. Ito, A. Saito, and T. Nishizeki. Secret sharing schemes realizing general access structures. In *Globecom*, pages 99–102, 1987.

[35] E. Kadric, P. Gurniak, and A. DeHon. Accurate parallel floating-point accumulation. In *21st IEEE Symp. on Computer Arithmetic (ARITH)*, pages 153–162, April 2013.

[36] L. Kamm and J. Willemson. Secure floating point arithmetic and private satellite collision analysis. *Int. Journal of Information Security*, 14(6):531–548, 2015.

[37] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley, 1997.

[38] H. Leuprecht and W. Oberaigner. Parallel algorithms for the rounding exact summation of floating point numbers. *Computing*, 28(2):89–104, 1982.

[39] M. A. Malcolm. On accurate floating-point summation. *Communications of the ACM*, 14(11):731–736, Nov. 1971.

[40] P. Mohassel and P. Rindal. ABY3: A mixed protocol framework for machine learning. In *ACM Conference on Computer and Communications Security (CCS)*, pages 35–52, 2018.

[41] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Springer, 2009.

[42] R. M. Neal. Fast exact summation using small and large superaccumulators. *arXiv ePrint*, abs/1505.05571, 2015.

[43] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In *10th IEEE Symp. on Computer Arithmetic (ARITH)*, pages 132–143, Jun 1991.

[44] D. Rathee, A. Bhattacharya, R. Sharma, D. Gupta, N. Chandran, and A. Rastogi. Secfloat: Accurate floating-point meets secure 2-party computation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1553–1553, 2022.

[45] J. Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[46] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part i: Faithful rounding. *SIAM J. on Scientific Computing*, 31(1):189–224, 2008.

[47] K. Sasaki and K. Nuida. Efficiency and accuracy improvements of secure floating-point addition over secret sharing. In *Int. Workshop on Security*, pages 77–94, 2020.

[48] A. Shamir. How to share a secret. *Comm. of the ACM*, 22(11):612–613, 1979.

[49] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[50] M. Tommila. Apfloat for Java. http://www.apfloat.org/apfloat_java/.

[51] L.-K. Wang, C. Tsen, M. J. Schulte, and D. Jhalani. Benchmarks and performance analysis of decimal floating-point applications. In *International Conference on Computer Design*, pages 164–170, 2007.

[52] Y.-K. Zhu and W. B. Hayes. Correct rounding and a hybrid approach to exact floating-point summation. *SIAM J. on Scientific Computing*, 31(4):2981–3001, 2009.

[53] Y.-K. Zhu and W. B. Hayes. Algorithm 908: Online Exact Summation of Floating-Point Streams. *ACM Transactions on Mathematical Software*, pages 1–13, 2010.