Hybrid Model of Fixed and Floating Point Numbers in Secure Multiparty Computations

Toomas Krips[2,3] and Jan Willemson[1,3]

[1] Cybernetica, Ülikooli 2, Tartu, Estonia
janwil@cyber.ee
[2] Institute of Computer Science, University of Tartu, Liivi 2, Tartu, Estonia
[3] STACC, Ülikooli 2, Tartu, Estonia
toomaskrips@gmail.com

Abstract. This paper develops a new hybrid model of floating point numbers suitable for operations in secure multi-party computations. The basic idea is to consider the significand of the floating point number as a fixed point number and implement elementary function applications separately of the significand. This gives the greatest performance gain for the power functions (e.g. inverse and square root), with computation speeds improving up to 18 times in certain configurations. Also other functions (like exponent and Gaussian error function) allow for the corresponding optimisation.

We have proposed new polynomials for approximation, and implemented and benchmarked all our algorithms on the Sharemind secure multi-party computation framework.

1 Introduction

Our contemporary society is growing more and more dependent on high-speed, high-volume data access. On one hand, such an access allows for developing novel applications providing services that were unimaginable just a decade ago. On the other hand, constant data flow and its automatic processing mechanisms are rising new security concerns every day.

In order to profit from the available data, but at the same time provide privacy protection for citizens, *privacy-preserving data analysis* (PPDA) mechanisms need to be applied. There exist numerous well-established statistical and data mining methods for data analysis. However, adding privacy-preservation features to them is far from being trivial. Many data processing primitives assume access to micro-data records, e.g. for joining different tables or even something as simple as sorting. There exist different methods for partial pre-aggregation and perturbation like k-anonymity [18, 20] and ℓ-diversity [17], but they reduce the precision of the dataset, and consequently decrease data utility.

Another approach is to tackle the PPDA problem from the privacy and cryptography point of view. Unfortunately, classical encryption methods (like block and stream ciphers) are meant only to scramble data and do not support meaningful computations on the plaintexts. More advanced methods like homomorphic encryption and searchable encryption [4] support some limited set of operations insufficient for the fully-featured statistical data analysis. There also exist methods for fully homomorphic encryption, but they are currently too inefficient to allow for analysis of a dataset of even a remotely useful size [9, 10].

Currently, one of the most promising techniques for cryptography-based PPDA is based on secret sharing and multi-party computations (SMC). There exist several frameworks allowing to work on relatively large amounts of secret-shared micro-data [2, 21]. In order to obtain the homomorphic behavior needed for Turing-completeness, they work over some algebraic structure (typically, a finite ring or field). However, to use the full variety of existing statistical tools, computations over real numbers are needed. Recently, several implementations of real-number arithmetic (both fixed and floating point) have emerged on top of SMC frameworks. While fixed point arithmetic is faster, floating point operations provide greater precision and flexibility. The focus of this paper is to explore the possibility of getting the best of both of the approaches and develop a hybrid fixed-floating point real numbers to be used with SMC applications.

2 Previous Work

Catrina and Saxena developed secure multiparty arithmetic on fixed-point numbers in [7], and their framework was extended with various computational primitives (like inversion and square root) in [7] and [15]. This fixed-point approach has been used to solve linear programming problems with applications in secure supply chain management [6, 12]. However, fixed point numbers provide only a limited amount of flexibility in computations, since they can represent values only in a small interval with a predetermined precision. Dahl *et al.* [8] use an approach that is rather close to fixed-point numbers to perform secure two-party integer division. They also use Taylor series to estimate $\frac{1}{x}$.

In order to access the full power of numerical methods, one needs an implementation of floating point arithmetic. This has been done by three groups of authors, Aliasgari *et al.* [1], Liu *et al.* [16], and Kamm and Willemson [11]. All these approaches follow the same basic pattern – the floating point number x is represented as $x = s \cdot f \cdot 2^e$, where s is the sign, f is the significand, and e is the exponent (possibly adjusted by a bias to keep the exponent positive). Additionally, Aliasgari *et al.* add a term to mark that the value of the floating point number is zero. Then all the authors proceed to build elementary operations of addition and multiplication, followed by some selection of more complicated functions.

Liu *et al.* [16] consider two-party additive secret sharing over a ring $\mathbb{Z}_N$ and only develop addition, subtraction, multiplication and division. Aliasgari *et al.* [1], use a threshold (t, n)-secret-sharing over a finite field and also develop several elementary functions such as logarithm, square root and exponentiation of floating-point numbers. All their elementary function implementations use different methods – square root is computed iteratively, logarithm is computed using a Taylor series and in order to compute the exponent, several *ad hoc* techniques are applied.

The research of Kamm and Willemson is motivated by a specific application scenario – satellite collision analysis [11]. In order to implement it, they need several elementary functions like inversion, square root, exponent and Gaussian error function. The authors develop a generic polynomial evaluation framework and use both Taylor and Chebyshev polynomials to get the respective numerical approximations.

2.1 Our Contribution

When evaluating elementary functions, both [1] and [11] use basic floating point operations as monolithic. However, this is not necessarily optimal, since oblivious floating point addition is a very expensive operation due to the need to align the points of the addends in an oblivious fashion. Fixed-point addition at the same time is a local (i.e. essentially free) operation, if an additively homomorphic secret sharing scheme is used. Hence, we may gain speedup in computation times if we are able to perform parts of the computations in the fixed-point representation. For example, in order to compute power functions (like inversion or square root), we can run the computations separately on the significand and exponent parts, but the significand is essentially a fixed-point number. Proposing, implementing and benchmarking this optimisation is the main contribution of this paper. We also propose new polynomials for various elementary functions to provide better precision-speed trade-offs.

Due to space restrictions, some of the technical details (most notably the particular polynomials) are omitted and are available in the full version of the paper [13].

3 Preliminaries

In the rest of the paper, we will assume a secret sharing scheme involving M parties $P_1, \ldots, P_M$. To share a value x belonging to ring (or field) $\mathbb{Z}_r$, it is split into M values $x_1, \ldots, x_M \in \mathbb{Z}_r$, and the share x_i is given to the party P_i ($i = 1, \ldots, M$). The secret shared vector $(x_1, \ldots, x_M)$ will be denoted as $[\![x]\!]$.

We will also assume that the secret sharing scheme is linear, implying that adding two shared values and multiplying a shared value by a scalar may be implemented component-wise, and hence require no communication between the computing parties. This is essential, since the running times of majority of SMC applications are dominated by the network communication. Note that many of the classical secret sharing schemes (like Shamir or additive scheme) are linear.

We will assume availability of the following elementary operations.

- Addition of two secret-shared values $[\![x]\!]$ and $[\![y]\!]$ denoted as $[\![x]\!] + [\![y]\!]$. Due to linearity, this evaluates to $[\![x + y]\!]$.
- Multiplication of a secret shared value $[\![x]\!]$ by a scalar $c \in \mathbb{Z}_r$ denoted as $c \cdot [\![x]\!]$. Due to linearity, this evaluates to $[\![c \cdot x]\!]$.
- Multiplication of two secret-shared values $[\![x]\!]$ and $[\![y]\!]$ denoted as $[\![x]\!] \cdot [\![y]\!]$. Unlike the two previous protocols, this one requires network communication to evaluate $[\![x \cdot y]\!]$.
- $\mathsf{PublicBitShiftRightProtocol}([\![x]\!], k)$. Takes a secret shared value $[\![x]\!]$ and a public integer k and outputs $[\![x \gg k]\!]$ where $x \gg k$ is equal to x shifted right by k bits. $x \gg k$ is equal to $\frac{x}{2^k}$ rounded down.
- $\mathsf{LTEProtocol}([\![x]\!], [\![y]\!])$. Gets two secret-shared values $[\![x]\!]$ and $[\![y]\!]$ as inputs and outputs a secret-shared bit $[\![b]\!]$. The bit b is set to 1 if $x \le y$ (interpreted as integers); otherwise, b is set to 0.
- $\mathsf{ObliviousChoiceProtocol}([\![b]\!], [\![x]\!], [\![y]\!])$. Gets a secret-shared bit b and two values $[\![x]\!]$ and $[\![y]\!]$ as inputs. If $b = 1$, the output will be set to $[\![x]\!]$, and if $b = 0$, it will be set to $[\![y]\!]$.
- $\mathsf{ConvertToBoolean}([\![x]\!])$.Takes in a secret-shared value $[\![x]\!]$ where x is equal to either 0 or 1, and converts it to the corresponding boolean value shared over $\mathbb{Z}_2$.
- $\mathsf{ConvertBoolToInt}([\![b]\!])$. Takes in a bit $[\![b]\!]$ secret-shared over $\mathbb{Z}_2$ and outputs a value $[\![x]\!]$ secret-shared over $\mathbb{Z}_r$, where x is equal to b as an integer.
- $\mathsf{GeneralizedObliviousChoice}([\![x_1]\!], \ldots, [\![x_k]\!], [\![\ell]\!])$. Takes an array of secret integers $[\![x_1]\!], \ldots, [\![x_k]\!]$ and a secret index $[\![\ell]\!]$ where $\ell \in [1, k]$, and outputs the shared integer $[\![x_\ell]\!]$.

- $\mathsf{BitExtraction}(\llbracket x \rrbracket)$. Takes in a secret integer $\llbracket x \rrbracket$ and outputs the vector of n secret values $\{\llbracket u_i \rrbracket\}_{i=0}^{n-1}$ where each $u_i \in \{0,1\}$ and $u_{n-1}u_{n-2}\dots u_0$ is the bitwise representation of $\llbracket x \rrbracket$.
- $\mathsf{PrivateBitShiftRightProtocol}(\llbracket x \rrbracket, \llbracket k \rrbracket)$ Takes a secret value $\llbracket x \rrbracket$ and a secret integer $\llbracket k \rrbracket$ and outputs $\llbracket x \gg k \rrbracket$ where $x \gg n$ is equal to x shifted right by k bits. When we apply this protocol to an n-bit secret integer $\llbracket x \rrbracket$ and k is not among $0, \dots, n-1$, the result will be $\llbracket 0 \rrbracket$.
- $\mathsf{ConvertToFloat}(\llbracket x \rrbracket)$ Takes a secret integer $\llbracket x \rrbracket$ and outputs a floating point number $\llbracket N \rrbracket = (\llbracket s_x \rrbracket, \llbracket E_x \rrbracket, \llbracket f_x \rrbracket)$ that is approximately equal to that integer.

Implementation details of these elementary operations depend on the underlying SMC platform. The respective specifications for Sharemind SMC engine and the complexities of some of the protocols can be found in [2,3,14].

4 Fixed-point Numbers

Our fixed-point arithmetic follows the framework of Catrina and Saxena [7], (for example, our multiplication protocol is based on that paper) but has several simplifications allowing for a more efficient software implementation.

First, instead of a finite field, we will be using a ring $\mathbb{Z}_{2^n}$ for embedding the fixed-point representations. Typically this ring will be $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$, since arithmetic in these rings is readily available in modern computer architectures. What we will lose is the possibility of using secret sharing over fields (including the popular Shamir's scheme [19]). Since our implementation will be based on Sharemind SMC engine [2], this is fine and we can use additive secret sharing instead.

The second simplification is made possible by our specific application of fixed-point numbers. The essential block we will need to build is polynomial evaluation on non-negative fixed point numbers (e.g. significands of floats). Even though we will occasionally need to cope with negative values, we will only represent non-negative fixed-point numbers. Besides the ring $\mathbb{Z}_{2^n}$ of n-bit integers, we will also fix the number m of bits we will interpret as the fractional part. We will consider the ring elements as unsigned, hence they run over the range $[0, 2^n - 1]$. We let the element $x \in \mathbb{Z}_{2^n}$ represent the fixed point number $x \cdot 2^{-m}$. Hence, the range of fixed point numbers we will be able to represent is $[0, 2^{n-m} - 2^{-m}]$, with granularity 2^{-m}. We will assume that all the fixed-point numbers we work on will be among these numbers. If we have to use some fractional number that cannot be represented in this way, we will automatically use the smallest representable fixed-point number that is greater than the number instead of this number.

We will use the following notation for fixed-point numbers. $\widetilde{x}$ denotes a fixed-point number, while x denotes the integer value we use to store $\widetilde{x}$ — namely, $\widetilde{x} \cdot 2^m$. Thus, when we have introduced some integer x, we have also defined the fixed-point number $\widetilde{x} = x \cdot 2^{-m}$ that it represents and vice versa. Likewise, when we want to denote a secret fixed-point number, we will write $[\![\widetilde{x}]\!]$ — this will be stored as a secret integer $[\![x]\!]$ where $x = \widetilde{x} \cdot 2^m$.

We will also need to denote numbers that are, in essence, public signed real numbers. For that, we will use the notation $s\widetilde{c}$ where $\widetilde{c}$ is the fixed-point number that denotes the absolute value of the real number and $s \in \{-1, 1\}$ is the sign of the real number.

4.1 Basic Operations on Fixed-point Numbers

We will now introduce the operations of addition and subtraction of two secret fixed-point numbers, multiplication of a secret fixed-point number and a public fixed point number and multiplication of two secret fixed-point numbers.

Addition of two secret fixed-point numbers $[\![\widetilde{x}]\!]$ and $[\![\widetilde{y}]\!]$ is free in terms of network communication, since this addition can be implemented by adding the representing values shared as the ring elements. Indeed, the sum of $\widetilde{x} = x \cdot 2^{-m}$ and $\widetilde{y} = y \cdot 2^{-m}$ is $(x + y) \cdot 2^{-m} = \widetilde{x + y}$. Hence we can compute $[\![\widetilde{x}]\!] + [\![\widetilde{y}]\!] = [\![\widetilde{x + y}]\!]$ just by adding the shares locally. The addition of the representatives takes place modulo 2^n and is unprotected against the overflow since checking whether the sum is too big would either leak information or would be expensive. Likewise, subtraction of two secret fixed-point numbers $[\![\widetilde{x}]\!]$ and $[\![\widetilde{y}]\!]$ is free in terms of network communication and can be implemented by subtracting the representing values shared as the ring elements. $[\![\widetilde{x - y}]\!]$ can be computed as $[\![\widetilde{x}]\!] - [\![\widetilde{y}]\!]$. The subtraction operation is also unprotected against going out of the range of the fixed-point numbers that we can represent and thus must be used only when it is known that $x \geq y$.

However, multiplication of a secret fixed-point number by other fixed point numbers, whether public or secret, is not free. Consider first multiplication of a public fixed-point number $\widetilde{a} = a \cdot 2^{-m}$ by a secret fixed-point number $[\![\widetilde{x}]\!] = [\![x]\!] \cdot 2^{-m}$. We need to calculate $[\![\widetilde{y}]\!]$ as the product of $\widetilde{a} = a \cdot 2^{-m}$ and $[\![\widetilde{x}]\!] = [\![x]\!] \cdot 2^{-m}$ where x is secret. Since we keep data as a and $[\![x]\!]$, we shall perform this computation as $a \cdot [\![x]\!] = \widetilde{a}2^m \cdot [\![\widetilde{x}]\!]2^m$. However, if we do this multiplication in $\mathbb{Z}_{2^n}$, then we risk losing the most significant bits, since the product $\widetilde{a}2^m\widetilde{x}2^m$ might be greater than 2^n.

In order to solve this problem, we convert a and $[\![x]\!]$ to $\mathbb{Z}_{2^{2n}}$ and compute the product in $\mathbb{Z}_{2^{2n}}$. Then we shift the product to the right by m bits and convert the number back to $\mathbb{Z}_{2^n}$, since the secret result y should be $\widetilde{a}[\![\widetilde{x}]\!] \cdot 2^m$, not $\widetilde{a}[\![\widetilde{x}]\!] \cdot 2^{2m}$. We assume that the product is in the range of the fixed-point numbers we can represent. We do not perform any checks to see that the multiplicands or the product are in the correct range, as this could leak information about the secret data, but instead assume that the user will adequately choose the input.After computing $a \cdot [\![x]\!] = \widetilde{a}2^m \cdot [\![\widetilde{x}]\!]2^m$ in $\mathbb{Z}_{2^{2n}}$, we note that the result should be $a \cdot [\![\widetilde{x}]\!]2^m$ and thus we need to divide the result by 2^m. The cheapest way to do this is shifting the numbers to the right by m bits.

There are two ways for doing that. The first one is using the existing protocol $\mathsf{PublicBitShiftRightProtocol}([\![y]\!], m)$ for shifting bits to the right. This protocol is not free, but gives the best possible result that can be represented with a given granularity and is guaranteed to give the correct result.The second one is to shift y_i to the right by m bits for every party P_i. This is free, but may be slightly inaccurate. Due to loss of the carry in the lowest bits we risk that the result might be smaller than the real product would be by at most $(M-1)\cdot 2^{-m}$. In most cases, this is an acceptable error. The only case where this error is significant is when our result should be among $[\![\widetilde{0}]\!], [\![\widetilde{2^{-m}}]\!], \ldots, (M-1)[\![\widetilde{2^{-m}}]\!]$ which could then be changed into one of $[\![\widetilde{2^{n-m} - (M-1)2^{-m}}]\!], \ldots, [\![\widetilde{2^{n-m} - 2^{-m}}]\!]$. To avoid this underflow, we add $[\![\widetilde{(M-1)2^{-m}}]\!]$ to the product after shifting. Note that now a symmetric problem where the shifted result should be among the numbers $[\![\widetilde{2^{n-m} - (M-1)2^{-m}}]\!], \ldots, [\![\widetilde{2^{n-m} - 2 \cdot 2^{-m}}]\!]$ or $[\![\widetilde{2^{n-m} - 2^{-m}}]\!]$ but would now be changed to one of $[\![\widetilde{0}]\!], [\![\widetilde{2^{-m}}]\!], \ldots, (M-1)[\![\widetilde{2^{-m}}]\!]$ could happen. However, we assume that the user would choose such inputs that these numbers would not arise as the products of any two multiplicands similarly as they would not multiply any two fixed-point numbers so that the product would be greater than $\widetilde{2^{n-m} - 2^{-m}}$. Now the user should not multiply any two fixed-point numbers so that the product would be greater than $\widetilde{2^{n-m} - M2^{-m}}$. Since M is usually a small number, this additional constraint does not practically affect computation.

The multiplication of two secret fixed-point numbers is similar. More specifically, to multiply two secret fixed-point numbers $[\![\widetilde{x}]\!]$ and $[\![\widetilde{y}]\!]$, we first convert $[\![x]\!]$ and $[\![y]\!]$ to $\mathbb{Z}_{2^{2n}}$ and then compute the product $[\![x]\!] \cdot [\![y]\!] =$

$\llbracket\widetilde{x}\rrbracket \cdot \llbracket\widetilde{y}\rrbracket 2^{2m} = \llbracket\widetilde{xy}\rrbracket \cdot 2^{2m} = \llbracket xy\rrbracket \cdot 2^m$ there. Then we shift $\llbracket xy\rrbracket \cdot 2^m$ to the right by m bits and add $\llbracket\widetilde{(M-1)2^{-m}}\rrbracket$ so that the result would be correct. After that we convert the result back to $\mathbb{Z}_{2^n}$ so that the product would be in the same ring as the multiplicands. We denote this operation by $\llbracket\widetilde{x}\rrbracket \cdot \llbracket\widetilde{y}\rrbracket$.

4.2 Polynomial Evaluation

```
Data: $\llbracket\widetilde{x}\rrbracket, m, n, s_i\{\widetilde{c_i}\}_{i=0}^{k}$
Result: Takes in a a secret fixed point number $\llbracket\widetilde{x}\rrbracket$, the radix-point $m$, the
        number of bits of the fixed-point number $n$ and the coefficients
        $s_i\{\widetilde{c_i}\}_{i=0}^{k}$ for the approximation polynomial. Outputs a secret
        fixed-point number $\llbracket\widetilde{y}\rrbracket$ that is the value of the approximation
        polynomial at point $x$.
$\llbracket\widetilde{x^1}\rrbracket \leftarrow \llbracket\widetilde{x}\rrbracket$
for $j \leftarrow 0$ to $\lceil \log_2(k) \rceil$ do
    for $i \leftarrow 1$ to $2^j$ do in parallel
        $\llbracket\widetilde{x^{i+2^j}}\rrbracket \leftarrow \llbracket\widetilde{x^{2^j}}\rrbracket \cdot \llbracket\widetilde{x^i}\rrbracket$
    end
end
$\llbracket\widetilde{y_0}\rrbracket \leftarrow \mathsf{Share}(\widetilde{c_0})$
for $i \leftarrow 1$ to $k$ do in parallel
    $\llbracket\widetilde{y_i}\rrbracket \leftarrow \widetilde{c_i} \cdot \llbracket\widetilde{x^i}\rrbracket$
end
$\llbracket\widetilde{y'}\rrbracket, \llbracket\widetilde{y''}\rrbracket \leftarrow \llbracket\widetilde{0}\rrbracket$
for $i \leftarrow 0$ to $k$ do in parallel
    if $s_i == 1$ then
        $\llbracket\widetilde{y'}\rrbracket += \llbracket\widetilde{y_i}\rrbracket$
    end
    if $s_i == -1$ then
        $\llbracket\widetilde{y''}\rrbracket += \llbracket\widetilde{y_i}\rrbracket$
    end
end
$\llbracket\widetilde{y}\rrbracket \leftarrow \llbracket\widetilde{y'}\rrbracket - \llbracket\widetilde{y''}\rrbracket$
return $\llbracket\widetilde{y}\rrbracket$
```

Algorithm 1: Computation of a polynomial on fixed-point numbers.

We will now present Algorithm 1 for evaluating polynomials with given coefficients. It is based on the respective algorithm described in [11] in the sense that the operations performed are the same but use fixed-point numbers instead of floating-point numbers. It takes in public signed coefficients $\{s_i\widetilde{c_i}\}_{i=0}^k$ and a secret fixed-point number $\llbracket\widetilde{x}\rrbracket$, and outputs $\llbracket\widetilde{y}\rrbracket = \sum_{i=0}^k s_i\widetilde{c_i}\cdot\llbracket\widetilde{x^k}\rrbracket$. Here $s_i \in \{-1, 1\}$. We will now describe the general strategy for that.

First we need to evaluate $\llbracket\widetilde{x^2}\rrbracket, \llbracket\widetilde{x^3}\rrbracket, \ldots, \llbracket\widetilde{x^k}\rrbracket$. It is trivial to do this with $k-1$ rounds of multiplications, however, we shall do it in $\lceil\log k\rceil$ rounds. Every round we compute the values $\llbracket\widetilde{x^{2^i+1}}\rrbracket, \llbracket\widetilde{x^{2^i+2}}\rrbracket, \ldots, \llbracket\widetilde{x^{2^{i+1}}}\rrbracket$ by multiplying $\llbracket\widetilde{x^{2^i}}\rrbracket$ with $\llbracket\widetilde{x^1}\rrbracket, \llbracket\widetilde{x^2}\rrbracket, \ldots, \llbracket\widetilde{x^{2^i}}\rrbracket$, respectively. (line 4)

Following that, on line 9 we can multiply the powers of x with the respective coefficients $\widetilde{c_i}$ with one round of multiplication, obtaining the values $\llbracket\widetilde{c_1x}\rrbracket, \llbracket\widetilde{c_2x^2}\rrbracket, \ldots, \llbracket\widetilde{c_kx^k}\rrbracket$. We also set $\llbracket\widetilde{c_0x^0}\rrbracket$ to the tuple of shares $(2^m \cdot c_0, 0, \ldots, 0)$. After that we can compute the sums $\llbracket\widetilde{\sum_{s_i=1} c_ix^i}\rrbracket = \sum_{s_i=1}\llbracket\widetilde{c_ix^i}\rrbracket$ and $\llbracket\widetilde{\sum_{s_i=-1} c_ix^i}\rrbracket = \sum_{s_i=-1}\llbracket\widetilde{c_ix^i}\rrbracket$ locally, respectively, on lines 14 and 17 and find the final result $\llbracket\widetilde{y}\rrbracket = \llbracket\widetilde{\sum_{s_i=1} c_ix^i}\rrbracket - \llbracket\widetilde{\sum_{s_i=-1} c_ix^i}\rrbracket$, which is also a local operation.

For every function, we face the question of which polynomial to use. Generally we have preferred using Chebyshev polynomials, to avoid the Runge phenomenon. For error function, we used Taylor series. However, sometimes large coefficients of Chebyshev polynomials can cause problems, such as making the result less accurate when the coefficients are very big. The reason for this is that to be able to represent large coefficients, the radix-point must be smaller and thus computing x^i will be more inaccurate for higher powers. We need to find the optimal place for the radix-point for each function. We also note that we will use this algorithm only so that it will output positive fixed-point numbers.

5 Hybrid Versions of Selected Functions

We have used the hybrid techniques to efficiently evaluate the square root, inverse, exponential and the Gaussian error function.

Our floating-point number representation is similar to the one from [11]. A floating-point number N consists of sign s, exponent E and significand f where $N = (-1)^{1-s} \cdot f \cdot 2^{E-q}$. Here q is a fixed number called the bias that is used for making the representation of the exponent non-negative. We require that if $N \neq 0$, the significand f would be normalised — i.e. f is represented by a number in $[2^{n-1}, 2^n - 1]$. If $N = 0$, then $f = 0$ and $E = 0$. If N is secret, then it means that the sign, significand and exponent are all independently secret-shared. We denote it with $\llbracket N\rrbracket = (\llbracket s\rrbracket, \llbracket E\rrbracket, \llbracket f\rrbracket)$.

Kamm and Willemson [11] present algorithms for computing the sum and product of two secret floating point numbers, and use these operations to implement polynomial approximations. However, the resulting routines are rather slow. Notably, computing the sum of two floating-point numbers is slower than computing the product. The basic structure of our function implementations is still inspired by [11].

The main improvement of the current paper is converting the significand of the floating-point number to a fixed-point number and then performing polynomial approximation in fixed-point format. The basic algorithm for polynomial evaluation was described in Algorithm 1. However, some extra corrections are needed after converting the result back into floating-point form. The general approach that we use for square root and inverse can easily be generalised for other power functions since for them we can work separately with the significand and the exponent. In order to use this approach for other functions (such as the exponential or the error function) work must be done to tailor specific arrangements for computing these functions in such a way.

5.1 Conversion from Fixed-point Number to Floating-point Number and Correction of Fixed-point Numbers

In three out of our four functions, when we evaluate the polynomial on some fixed-point number $[\![\widetilde{x}]\!]$ where $\widetilde{x} \in [2^v, 2^{v+1})$, and we get $[\![\widetilde{y}]\!]$ as the output, where $\widetilde{y}$ should be in $[2^t, 2^{t+1})$ for some t that depends on the function. For example, for inverse, if the input $[\![\widetilde{x}]\!]$ is in $[0.5, 1)$, then the output should be approximately in $[1, 2)$.

However, due to inaccuracies coming from roundings and the error of the polynomial, the result might be out of that range— it might also be in $[0, 2^t)$ or $[2^{t+1}, 2^{t+2})$. If that should happen we will use the protocol $\mathsf{Correction}([\![\widetilde{y}]\!], t, m, n, b_0, b_1)$ to get a result that is in the correct range and is not less accurate. Here b_0 and b_1 are public boolean flags that are set to 1 if the result may be in $[0, 2^t)$ or $[2^{t+1}, 2^{t+2})$, respectively. We omitted this algorithm from this version of the paper due to size constraints. It can be read in the full version of the paper.

This algorithm is necessary in several cases for converting a fixed-point number back to the significand of a floating-point number. If this sort of protocol is not performed and we mistakenly assume that the fixed-point number $\widetilde{x}$ that we got as a result is in some $[2^t, 2^{t+1})$, and thus we set the result to $[\![N]\!] = ([\![s]\!], 2^{n-m-t-1} \cdot [\![x]\!], [\![t + q]\!])$, then it might happen that the floating-point number N is not normalised.

We also omitted an algorithm $\mathsf{FixToFloatConversion}(\llbracket\widetilde{x}\rrbracket, t, m, n)$ that is used for converting a positive fixed-point number $\llbracket\widetilde{x}\rrbracket$ to a floating-point number if we know that $\widetilde{x} \in [2^{t-1}, 2^{t+1})$. If $\widetilde{x} \in [2^{t-1}, 2^{t})$, then our result should be $\llbracket N_1\rrbracket = (\llbracket s\rrbracket, \llbracket E\rrbracket, \llbracket f\rrbracket) = (\llbracket 1\rrbracket, \llbracket t+q\rrbracket, \llbracket\widetilde{y}\rrbracket \cdot 2^{n-t})$. If $\widetilde{x} \in [2^{t}, 2^{t+1})$, then our result should be $\llbracket N_2\rrbracket = (\llbracket s\rrbracket, \llbracket E\rrbracket, \llbracket f\rrbracket) = (\llbracket 1\rrbracket, \llbracket t+q+1\rrbracket, \llbracket\widetilde{y}\rrbracket \cdot 2^{n-t-1})$.

5.2 Inverse

We will describe Algorithm 2 for computing the inverse of a floating-point number $\llbracket N\rrbracket = (\llbracket s\rrbracket, \llbracket E\rrbracket, \llbracket f\rrbracket)$ in our setting.

```
   Data: $\llbracket N\rrbracket = (\llbracket s\rrbracket, \llbracket E\rrbracket, \llbracket f\rrbracket), q, m, \{s_i\widetilde{c_i}\}_{i=0}^{k}, n$
   Result: Takes in a a secret floating point number $\llbracket N\rrbracket = (\llbracket s\rrbracket, \llbracket E\rrbracket, \llbracket f\rrbracket)$, the
           bias of the exponent $q$ and the radix-point of the corresponding
           fixed-point number $m$, Chebyshev coefficients $\{\widetilde{c_i}\}_{i=0}^{k}$ for computing
           the fixed-point polynomial and the number of bits of the fixed-point
           number $n$. Outputs a secret floating-point number that is
           approximately equal to the inverse of $N$.
1  $\llbracket f'\rrbracket \leftarrow \mathsf{PublicBitShiftRightProtocol}(\llbracket f\rrbracket, n-m)$
2  $\llbracket t\rrbracket \leftarrow \mathsf{FixInverseProtocol}(\llbracket\widetilde{f'}\rrbracket, \{s_i\widetilde{c_i}\}_{i=0}^{k}, m, n)$
3  $\llbracket t'\rrbracket \leftarrow \mathsf{Correction}(\llbracket\widetilde{t}\rrbracket, 0, m, n, 0, 1)$
4  $\llbracket t''\rrbracket \leftarrow \llbracket t'\rrbracket \cdot 2^{n-m-1}$
5  return $\llbracket N'\rrbracket = (\llbracket s\rrbracket, \llbracket 2q-E+1\rrbracket, \llbracket t''\rrbracket)$
```

Algorithm 2: Inverse of a floating point number.

First note that since inverse of zero is not defined, we can assume that the input is not zero and that thus the signicand is always normalised. Second, note that the significand $\llbracket f\rrbracket$ can now be considered a fixed-point number where $m = n$ as it represents a number in $[0.5, 1)$ but is stored as a shared value in $[2^{n-1}, 2^{n}-1]$. However, if the radix-point is so high, we can not perform most of the operations we need to, so on line 1 we shift the significand to the standard fixed-point format. We lose $n-m$ bits, but since the significand has more bits for its significand than the IEEE standard 754 for both single and double precision, the number of bits we have left is not less than the significand of the IEEE standard has. Let us denote the shifted significand with $\llbracket\widetilde{f'}\rrbracket$. Then, on line 2, we securely compute the number $\llbracket t\rrbracket$ so that $\widetilde{t}$ is the inverse of $\widetilde{f'}$ by using polynomial evaluation, as described in Algorithm 1.

The exact polynomials we used for the fixed point inversion can be found in the full version of the paper [13]. We will denote calling the

Algorithm 1 on value $[\![\widetilde{x}]\!]$ with the coefficients of that polynomial by $\mathsf{FixInverseProtocol}([\![\widetilde{x}]\!], \{s_i\widetilde{c_i}\}_{i=0}^{k}, m, n)$, where m is the position of the radix point and n is the number of bits in the fixed-point number and where $\{s_i\widetilde{c_i}\}_{i=0}^{k}$ refers to the signed coefficients of the polynomial. Calling $\mathsf{FixInverseProtocol}$ on $[\![\widetilde{x}]\!]$ gives us $\widetilde{t'}$.

Since $\widetilde{f'} \in [0.5, 1)$, we expect the result $\widetilde{t'}$ to be approximately in $(1, 2]$. However, since the polynomial has a small error, then the result might sometimes be slightly bigger than 2 and thus on line 3 we need to correct the result using the $\mathsf{Correction}$ algorithm with range parameter being 0.

Next we want to divide the result by two and then convert the fixed-point number back into the significand format. We can combine these two operations. The first one would require shifting to the right by one bit and the second one would require shifting to the left by $n - m$ bits. By combining, we just have to shift the result to the left by $n - m - 1$ bits, which is a free operation since it is equivalent with multiplying by 2^{n-m-1} which we do on line 4. The sign of the inverse is the same as the sign of N and the exponent should be the additive inverse of the original exponent, minus one to take into account the division by two that we did in the significand. However, we need to take into account that the bias is added to the exponent and thus the exponent of the result shall be $[\![-E + q + 1]\!]$. Thus we obtain Algorithm 2 for computing the inverse of a floating-point number.

5.3 Square Root

We will describe Algorithm 3 for computing the square root of a floating-point number in our setting. Note that since we assume that the sign is positive and thus ignore the sign, we will de facto compute the function $\sqrt{|x|}$. If the input is $-x$ for some non-negative x, then the output will be approximately $\sqrt{x}$.

First we shall describe the case where the input is not zero. We note that the significand $[\![f]\!]$ can be considered a fixed-point number where $m = n$ as it represents a number in $[0.5, 1)$ but is stored as a shared value in $[2^{n-1}, 2^n - 1]$. However, if the radix-point is so big, we can not perform most of the operations we need to, so on line 1, we shift the significand to the standard fixed-point format. Let us denote the shifted significand by $[\![\widetilde{f'}]\!]$. While computing the square root, it is natural to halve the exponent by shifting it to the right by one bit on line 3. However, the parity of that last bit may change the result $\frac{\sqrt{2}}{2}$ times and thus we have to remember the last bit before that on line 2 and later use it to perform an oblivious

```
Data: $[\![N]\!] = ([\![s]\!], [\![E]\!], [\![f]\!]), q, m, \{s_i\widetilde{c_i}\}_{i=0}^{k}, n$
Result: Takes in a a secret floating point number $[\![N]\!] = ([\![s]\!], [\![E]\!], [\![f]\!])$, the
        bias of the exponent $q$ and the radix-point of the corresponding
        fixed-point number $m$, Chebyshev coefficients $\{\widetilde{c_i}\}_{i=0}^{k}$ for computing
        the fix-point polynomial and the number of bits of the fixed-point
        number $n$. Outputs a secret floating-point number that is
        approximately equal to $\sqrt{N}$.
1 $[\![\widetilde{f'}]\!] \leftarrow$ PublicBitShiftRightProtocol($[\![f]\!], n-m$)
2 $[\![b]\!] \leftarrow [\![E]\!] \pmod 2$
3 $[\![E']\!] \leftarrow$ PublicBitShiftRightProtocol($[\![E]\!], 1$)
4 $[\![\widetilde{t_1}]\!] \leftarrow$ FixSquareRootProtocol($[\![\widetilde{f'}]\!], \{s_i\widetilde{c_i}\}_{i=0}^{k}, m, n$)
5 $[\![\widetilde{t_2}]\!] \leftarrow [\![\widetilde{t_1}]\!] \cdot \widetilde{\frac{\sqrt{2}}{2}}$
6 $[\![\widetilde{t_2'}]\!] \leftarrow$ Correction($[\![\widetilde{t_2}]\!], -1, m, n, 1, 0$)
7 $[\![t']\!] \leftarrow$ ObliviousChoiceProtocol($[\![b]\!], [\![\widetilde{t_1}]\!], [\![\widetilde{t_2'}]\!]$)
8 $[\![t'']\!] \leftarrow [\![t']\!] \ll (n-m)$
9 return $[\![N']\!] = ([\![1]\!], [\![E' + 1 + (q \gg 1)]\!], [\![t'']\!])$
```

Algorithm 3: Square root of a floating point number.

choice on line 7. Like in the case of the inverse, we use a Chebyshev polynomial on line 4 to find such $[\![\widetilde{t_1}]\!]$ that $\widetilde{t_1}$ is approximately equal to the square root of $\widetilde{f'}$. For that we compute the square root of $[\![\widetilde{f'}]\!]$ by using polynomial evaluation, as described in Algorithm 1.The exact polynomial for computing the fixed point square root can be found in the full version of the paper [13].

We will denote calling the function 1 on value $[\![\widetilde{x}]\!]$ with the coefficients of that polynomial with FixSquareRootProtocol($[\![\widetilde{x'}]\!], \{s_i\widetilde{c_i}\}_{i=0}^{k}, m, n$) where m is the position of the radix point and n is the number of bits in the fixed-point number and where $\{s_i\widetilde{c_i}\}_{i=0}^{k}$ refers to the signed coefficients of the polynomial. Calling FixSquareRootProtocol on $[\![\widetilde{x}]\!]$ gives us $[\![\widetilde{t_1}]\!]$.

Following that, on line 5 we multiply $[\![\widetilde{t_1}]\!]$ by $\widetilde{\frac{\sqrt{2}}{2}}$ —we then have the risk of $[\![\widetilde{t_1 \cdot \frac{\sqrt{2}}{2}}]\!]$ being slightly less than $\widetilde{0.5}$, thus we need to use the Correction with range parameter being -1 on line 6 to correct $[\![\widetilde{t_1 \cdot \frac{\sqrt{2}}{2}}]\!]$ into the range $[0.5, 1)$. Then, on line 7, we use the saved last bit of the exponent to perform an oblivious choice between $[\![\widetilde{t_1}]\!]$ and $[\![\widetilde{t_1 \cdot \frac{\sqrt{2}}{2}}]\!]$ and convert the result back into the significand format by shifting the result left by $n-m$ bits on the line 8. The latter operation may be implemented by multiplying the result by 2^{n-m} which is a local operation. The sign of a square root is always plus. We correct for the bias and rounding errors by adding $1 + (q \gg 1)$ to $[\![E']\!]$. The added 1 comes from the fact that the bias is odd and we lose 0.5 from the exponent twice when truncating q by a bit. Thus we obtain Algorithm 3 for computing the square root of a floating-point number. The algorithm also gives a correct result if the input is zero but the reasoning for this was omitted due to size constraints and can be read in the full version of the paper.

5.4 Exponent

```
Data: $[\![N]\!] = ([\![s]\!], [\![E]\!], [\![f]\!]), q, m, \{s_i\widetilde{c_i}\}_{i=0}^{k}, n$
Result: Takes in a a secret floating point number $[\![N]\!] = ([\![s]\!], [\![E]\!], [\![f]\!])$, the
        bias of the exponent $q$ and the radix-point of the corresponding
        fixed-point number $m$, coefficients $\{s_i\widetilde{c_i}\}_{i=0}^{k}$ for computing the
        fix-point polynomial and the number of bits of the fixed-point number
        $n$. Outputs a secret floating-point number that is approximately equal
        to $e^N$.
$[\![y]\!] = ([\![s_y]\!], [\![E_y]\!], [\![f_y]\!]) \leftarrow \log_2 e \cdot [\![N]\!]$
$[\![z]\!] \leftarrow \mathsf{PrivateBitShiftRightProtocol}([\![f_y]\!], [\![n - (E_y - q)]\!])$
$[\![[y]]\!] = ([\![s_{[y]}]\!], [\![E_{[y]}]\!], [\![f_{[y]}]\!]) \leftarrow \mathsf{ConvertToFloat}([\![z]\!])$
$[\![\{y\}]\!] = ([\![s_{\{y\}}]\!], [\![E_{\{y\}}]\!], [\![f_{\{y\}}]\!]) \leftarrow [\![y]\!] - [\![[y]]\!]$
$[\![\widetilde{w}]\!] \leftarrow \mathsf{PrivateBitShiftRightProtocol}([\![f_{\{y\}}]\!], [\![-E_{\{y\}} + q + n - m]\!])$
begin in parallel
    $[\![\widetilde{f'}]\!] \leftarrow \mathsf{FixPowerOfTwoProtocol}([\![\widetilde{w}]\!], \{s_i\widetilde{c_i}\}_{i=0}^{k}, m, n)$
    $[\![\widetilde{f''}]\!] \leftarrow \mathsf{FixPowerOfTwoProtocol}([\![\widetilde{1 - w}]\!], \{s_i\widetilde{c_i}\}_{i=0}^{k}, m, n)$
end
begin in parallel
    $[\![\widetilde{f'}]\!] \leftarrow \mathsf{Correction}([\![\widetilde{f'}]\!], 0, m, n, 1, 1)$
    $[\![\widetilde{f''}]\!] \leftarrow \mathsf{Correction}([\![\widetilde{f''}]\!], 0, m, n, 1, 1)$
end
$[\![2^{\{y'\}}]\!] = ([\![s_{2^{\{y'\}}}]\!], [\![E_{2^{\{y'\}}}]\!], [\![f_{2^{\{y'\}}}]\!]) \leftarrow ([\![1]\!], [\![q + 1]\!], [\![f' \cdot 2^{n-m-1}]\!]$
$[\![2^{\{y''\}}]\!] = ([\![s_{2^{\{y''\}}}]\!], [\![E_{2^{\{y''\}}}]\!], [\![f_{2^{\{y''\}}}]\!]) \leftarrow ([\![1]\!], [\![q + 1]\!], [\![f'' \cdot 2^{n-m-1}]\!]$
$[\![2^{[y']}]\!] = ([\![s_{2^{[y']}}]\!], [\![E_{2^{[y']}}]\!], [\![f_{2^{[y']}}]\!]) \leftarrow ([\![1]\!], [\![1 + q + z]\!], [\![100\ldots0]\!])$
$[\![2^{[y'']}]\!] = ([\![s_{2^{[y'']}}]\!], [\![E_{2^{[y'']}}]\!], [\![f_{2^{[y'']}}]\!]) \leftarrow ([\![1]\!], [\![q - z]\!], [\![100\ldots0]\!])$
$[\![b]\!] \leftarrow \mathsf{ConvertToBoolean}([\![s_y]\!])$
begin in parallel
    $[\![2^{\{y\}}]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![b]\!], [\![2^{\{y'\}}]\!], [\![2^{\{y''\}}]\!])$
    $[\![2^{[y]}]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![b]\!], [\![2^{[y']}]\!], [\![2^{[y'']}]\!])$
end
$[\![2^y]\!] = ([\![s_{2^y}]\!], [\![E_{2^y}]\!], [\![f_{2^y}]\!]) \leftarrow [\![2^{[y]}]\!] \cdot [\![2^{\{y\}}]\!]$
return $[\![N']\!] = [\![2^y]\!]$
```

Algorithm 4: Power of e of a floating point number.

We will describe Algorithm 4 for computing the exponent of a floating-point number in our setting.Given a secret floating-point number $[\![N]\!] = ([\![s]\!], [\![E]\!], [\![f]\!])$ we wish to compute $[\![e^N]\!] = [\![2^{\log_2 e \cdot N}]\!] = [\![2^y]\!]$ where $y := \log_2 e \cdot N$.

It is easier to compute a power of 2 in our setting than a power of e so thus on line 1 we first compute the floating-point number $[\![y]\!] = ([\![s_{\{y\}}]\!], [\![E_{\{y\}}]\!], [\![f_{\{y\}}]\!]) = [\![\log_2 e]\!] \cdot [\![N]\!]$. To compute $[\![2^y]\!]$, we split $[\![y]\!]$ into two parts — the integer part $[\![[y]]\!]$ and the fractional part $[\![\{y\}]\!]$. Note that the $n - (E_y - q)$ last bits of the s_y represent the fractional part of y and the rest of the bits represent the integer part, so it is equal to $2^{n-(E_y-q)}([y] + \{y\})$ Thus, on line 2 we privately shift $[\![f_y]\!]$ to the right by $[\![n - (E_y - q)]\!]$ bits to truncate the fractional part and divide by $2^{n-(E_y-q)}$ and thus obtain the integer part of $[\![y]\!]$ that we represent with $[\![z]\!]$. This, however, has integer type and we want to deduce it from a floating-point number so we need to call the ConvertToFloat method on line 3 to cast $[\![z]\!]$ into a floating point-number $[\![[y]]\!] = ([\![s_{[y]}]\!], [\![E_{[y]}]\!], [\![f_{[y]}]\!])$.

We find the fractional part $[\![\{y\}]\!] = ([\![s_{\{y\}}]\!], [\![E_{\{y\}}]\!], [\![f_{\{y\}}]\!])$ on line 4 by $[\![\{y\}]\!] = [\![y]\!] - [\![[y]]\!]$. Note that if $[y]$ is negative, then the fractional part will be in $[-1, 0)$, so we have to subtract 1 from $[\![[y]]\!]$ and add it to $[\![\{y\}]\!]$ in order for $\{y\}$ to be in $[0, 1)$. We shall do the next operations in both the positive and the negative case and use oblivious choice in the end to choose between them. Now we convert $[\![\{y\}]\!]$ to a fixed-point number $[\![\widetilde{w}]\!]$ by shifting $[\![f_{\{y\}}]\!]$ to the right by $[\![-E_{\{y\}} + q + n - m]\!]$ bits. The exact polynomial for computing the fixed point exponent can be found in the full version of the paper [13].

We will denote calling the Algorithm 1 on value $[\![\widetilde{x}]\!]$ with the coefficients of that polynomial by FixPowerOfTwoProtocol($[\![\widetilde{x}]\!], \{s_i \widetilde{c_i}\}_{i=0}^{k}, m, n$), where m is the position of the radix point and n is the number of bits in the fixed-point number and where $\{s_i \widetilde{c_i}\}_{i=0}^{k}$ refers to the signed coefficients of the polynomial and we call this function in parallel on lines 7 and 8 this polynomial on values $[\![\widetilde{w}]\!]$ and $[\![\widetilde{1 - w}]\!]$ and thus obtain $[\![\widetilde{f'}]\!]$ and $[\![\widetilde{f''}]\!]$ respectively and correct them with range parameter 0. We then initialize the possible values for $[\![2^{\{y\}}]\!]$, that is, $[\![2^{\{y'\}}]\!]$ and $[\![2^{\{y''\}}]\!]$ with signs 1, exponents $q + 1$ and significands that are equal to $[\![\widetilde{f'}]\!]$ and $[\![\widetilde{f''}]\!]$ that have been shifted by $n - m - 1$ bits to the left. Likewise, we initialize the possible values for $[\![2^{[y]}]\!]$, that is, $[\![2^{[y']}]\!]$ and $[\![2^{[y'']}]\!]$ with signs 1, exponents that are equal to $1 + q + z$ and $q - z$, respectively and significands $100\dots0 = 2^{n-1}$. $[\![2^{[y']}]\!]$ and $[\![2^{\{y'\}}]\!]$ are the correct values if the input was a positive number, i.e. $s_y = 1$ and $[\![2^{[y'']}]\!]$ and $[\![2^{\{y''\}}]\!]$ are the correct values if the input was a negative number i.e. $s_y = 0$. Thus we convert $[\![s_y]\!]$ to a boolean on line 18 and, in parallel, perform oblivious choice between $[\![2^{\{y'\}}]\!]$ and $[\![2^{\{y''\}}]\!]$ on line 20 and $[\![2^{[y']}]\!]$ and $[\![2^{[y'']}]\!]$ on line 21 to respectively obtain $[\![2^{\{y\}}]\!]$ and $[\![2^{[y]}]\!]$. We then multiply $[\![2^{\{y\}}]\!]$ and $[\![2^{[y]}]\!]$ together on line 23 to obtain the result. Thus we obtain the Algorithm 4 for computing the exponential function of a floating-point number. Note that the algorithm also works on inputs 0 and -0.

5.5 Error Function

Gaussian error function is defined by $\operatorname{erf} x = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$. It is a anti-symmetric function — i.e. $\operatorname{erf}(-x) = -\operatorname{erf}(x)$. Thus we can evaluate the function only depending on the exponent and the significand, and in the end, set the sign of the output to be the sign of the input. So, for the sake of simplicity, we will assume that our input is non-negative. However, since $\operatorname{erf}(a \cdot b)$ can not be easily computed from $\operatorname{erf} a$ and $\operatorname{erf} b$, we can not use the approach we used for inverse and square root. Computing the error function using approximation polynomials on significands does not seem possible, as we would have to be able to represent all numbers with fixed-point numbers of a good precision (conflicting goals) and also use very many approximation polynomials.

However, it turns out that we can bound the range of inputs in which case we have to compute the error function with a fixed-point polynomial. Namely, if x is close to 0 then $\operatorname{erf} x$ can be well approximated with $\frac{2}{\sqrt{\pi}}x$—observe that the McLaurin series of the error function is $\operatorname{erf} x = \frac{2}{\sqrt{\pi}}\sum_{i=0}^{\infty}\frac{(-1)^n}{n!(2n+1)}x^{2n+1}$ and note that $|\operatorname{erf} x - \frac{2}{\sqrt{\pi}}x| = |\frac{2}{\sqrt{\pi}}\sum_{i=1}^{\infty}\frac{(-1)^n x^{2n+1}}{n!(2n+1)}| < \frac{2x}{3\cdot\sqrt{\pi}}\sum_{i=1}^{\infty}x^{2n} = \frac{2}{3\cdot\sqrt{\pi}}\frac{x^3}{1-x^2}$.

If x is small enough, then $\frac{2}{3\cdot\sqrt{\pi}}\frac{x^3}{1-x^2}$ is negligible. On the other hand, $\operatorname{erf} x$ is a monotonously growing function that approaches 1 so we can approximate $\operatorname{erf}(x)$ with 1 for large enough x. In our approach, if $x \geq 4$, we set $\operatorname{erf} x = 1$. The error we make is at most $2 \cdot 10^{-8}$. Thus, we will only need to compute polynomial approximations for $x \in [2^{-w}, 2^2)$ where w is a previously fixed public parameter that depends on how precise we would like the algorithm to be.

Thus we need approximation polynomials for the range $[0, 4)$ only. We will use four approximation polynomials, p_0, p_1, p_2 and p_3 where $p_i(\widetilde{y}) \approx \operatorname{erf} y$ if $\widetilde{y} \in [i, i+1)$, where $i \in \{0, 1, 2, 3\}$. The exact polynomials p_1, p_2, p_3 and p_4 can be found in the full version of the paper [13].

We shall now describe Algorithm 5 for computing the error function of a floating-point number. First, we shall find the possible corresponding

```
Data: $\llbracket N \rrbracket = (\llbracket s \rrbracket, \llbracket E \rrbracket, \llbracket f \rrbracket), q, m,$
      $\{s_{i,0}\widetilde{c_{i,0}}\}_{i=0}^{l}, \{s_{i,1}\widetilde{c_{i,1}}\}_{i=0}^{l}, \{s_{i,2}\widetilde{c_{i,2}}\}_{i=0}^{l}, \{s_{i,3}\widetilde{c_{i,3}}\}_{i=0}^{l}, n, w$
Result: Takes in a a secret floating point number $\llbracket N \rrbracket = (\llbracket s \rrbracket, \llbracket E \rrbracket, \llbracket f \rrbracket)$, the
        bias of the exponent $q$ and the radix-point of the corresponding
        fixed-point number $m$, coefficients $\{s_{i,j}\widetilde{c_{i,j}}\}_{i=0}^{l}$ for computing the
        fixed-point values that are accurate in $[j, j+1)$ and an integer $w$ so
        that we evaluate the function with a polynomial, if $2^w \leq N < 4$.
        Outputs a secret floating-point number that is approximately equal to
        the error function of $N$.
for $k \leftarrow 0$ to $w$ do
    $shifts_k \leftarrow n - m + i - 2$
end
$\{\llbracket f_k \rrbracket\}_{k=0}^{w} \leftarrow$ PublicBitShiftRightProtocol($\llbracket f \rrbracket, \{shifts\}_{k=0}^{w})$)
for $k \leftarrow 1$ to $w$ do in parallel
    $\llbracket \widetilde{g_k} \rrbracket \leftarrow$ FixGaussianErrorFunction($\llbracket \widetilde{f_k} \rrbracket, m, n, \{s_{i,0}\widetilde{c_{i,0}}\}_{i=0}^{l}$)
    $\llbracket \widetilde{g_0} \rrbracket \leftarrow$ FixGaussianErrorFunction($\llbracket \widetilde{f_0} \rrbracket, m, n, \{s_{i,1}\widetilde{c_{i,1}}\}_{i=0}^{l}$)
    $\llbracket \widetilde{g_{-1,0}} \rrbracket \leftarrow$ FixGaussianErrorFunction($\llbracket \widetilde{f_0} \rrbracket, m, n, \{s_{i,2}\widetilde{c_{i,2}}\}_{i=0}^{l}$)
    $\llbracket \widetilde{g_{-1,1}} \rrbracket \leftarrow$ FixGaussianErrorFunction($\llbracket \widetilde{f_0} \rrbracket, m, n, \{s_{i,3}\widetilde{c_{i,3}}\}_{i=0}^{l}$)
end
$\{\llbracket u_i \rrbracket\}_{i=0}^{n} \leftarrow$ BitExtraction($\llbracket f \rrbracket$)
$\llbracket g_{-1} \rrbracket \leftarrow$ ObliviousChoiceProtocol($\llbracket u_m \rrbracket, \llbracket g_{-1,1} \rrbracket, \llbracket g_{-1,0} \rrbracket$)
$t_{-1} \leftarrow 0$
$t_0 \leftarrow 0$
for $k \leftarrow 1$ to $w$ do
    $t_k \leftarrow 2 - k$
end
for $k \leftarrow -1$ to $w$ do in parallel
    $\llbracket N_k \rrbracket = (\llbracket s_k \rrbracket, \llbracket E_k \rrbracket, \llbracket f_k \rrbracket) \leftarrow$ FixToFloatConversion($\llbracket \widetilde{g_k} \rrbracket, t_k, m, n$)
end
$\llbracket N_{-2} \rrbracket = (\llbracket s_{-2} \rrbracket, \llbracket E_{-2} \rrbracket, \llbracket f_{-2} \rrbracket) \leftarrow \frac{2}{\sqrt{\pi}} \cdot \llbracket N \rrbracket$
$\llbracket N_{w+1} \rrbracket = (\llbracket s_{w+1} \rrbracket, \llbracket E_{w+1} \rrbracket, \llbracket f_{w+1} \rrbracket) \leftarrow 1$
begin in parallel
    $b_0 \leftarrow$ LTEProtocol($\llbracket E \rrbracket, \llbracket q - w \rrbracket$)
    $b_1 \leftarrow$ LTEProtocol($\llbracket q + 3 \rrbracket, \llbracket E \rrbracket$)
end
$\llbracket E \rrbracket \leftarrow$ ObliviousChoiceProtocol($\llbracket b_0 \rrbracket, \llbracket q - w \rrbracket, \llbracket E \rrbracket$)
$\llbracket E \rrbracket \leftarrow$ ObliviousChoiceProtocol($\llbracket b_1 \rrbracket, \llbracket q + 3 \rrbracket, \llbracket E \rrbracket$)
$\llbracket E' \rrbracket \leftarrow$ GeneralizedObliviousChoice($\llbracket E_{-2} \rrbracket, \ldots, \llbracket E_{w+1} \rrbracket, \llbracket E - q \rrbracket$)
$\llbracket f' \rrbracket \leftarrow$ GeneralizedObliviousChoice($\llbracket f_{-2} \rrbracket, \ldots, \llbracket f_{w+1} \rrbracket, \llbracket E - q \rrbracket$)
return $\llbracket N' \rrbracket = (\llbracket s \rrbracket, \llbracket E' \rrbracket, \llbracket f' \rrbracket)$
```

Algorithm 5: Gaussian error function of a floating point number.

fixed-point numbers on which we compute our polynomial. We will, in parallel, on line 4, compute

$$\llbracket\widetilde{f_i}\rrbracket = \mathsf{PublicBitShiftRightProtocol}(\llbracket f\rrbracket, n - m + i - 2) \text{ for } i \in [0, w].$$

If $\llbracket f\rrbracket$ is the significand of $\llbracket x\rrbracket$ then $\llbracket\widetilde{f_i}\rrbracket \in [2^{-i}, 2^{-i+1})$ if $x \in [2^{-i}, 2^{-i+1})$.

Note that although we need polynomial approximation for values that are in $[2, 4)$, we did not compute any such fixed-point number $\widetilde{f_{-1}} = \mathsf{PublicBitShiftRightProtocol}(\llbracket f\rrbracket, n - m - 3)$ that is equal to x if $x \in [2, 4)$. Instead of computing the polynomial $\sum_{i=0}^{l} s_i a_i \cdot \widetilde{f_{-1}}^{i}$ we compute the polynomial $\sum_{i=0}^{l} s_i a_i 2^i \cdot \widetilde{f_0}^{i}$. Note that these two expressions are almost equivalent since $f_0 = f_{-1} \ll 1$. However, the latter is preferable since we will only be able to represent $\widetilde{f_{-1}}^{i}$ if i is very small and thus not be able to use good polynomials.

We now wish to compute values $\llbracket\widetilde{g_i}\rrbracket$ where $\widetilde{g_i} \approx \operatorname{erf} \widetilde{f_i}$ if $x \in [2^{-i}, 2^{-i+1})$ for $i \in [-1, w]$. For $i \in [1, w]$ we compute $\widetilde{g_i} = p_0(\widetilde{f_i})$ on line 6. For $i = 0$, we compute $\widetilde{g_0} = p_1(\widetilde{f_0})$ on line 7 For $i = -1$, we compute $\widetilde{g_{-1,0}}$ and $\widetilde{g_{-1,1}}$ on lines 8 and 9 by applying the modified versions of the polynomials p_2 and p_3 to $\widetilde{f_0}$, as described before. Note that these values are computed in parallel. Now we need to evaluate $\widetilde{g_{-1}}$ using oblivious choice so that if the result is in $[2, 4)$, $\widetilde{g_{-1}} = \widetilde{g_{-1,0}}$ if $f \leq 2^{n-1}$ and $\widetilde{g_{-1}} = \widetilde{g_{-1,1}}$ if $f > 2^{n-1}$. We note that whether $f \leq 2^{n-1}$ or not depends only on the last bit of f, thus we use the $\mathsf{BitExtract}$ protocol on $\llbracket f\rrbracket$ on line 11 to find that bit and use that to perform oblivious choice on line 12 between $\widetilde{g_{-1,0}}$ and $\widetilde{g_{-1,1}}$.

Note that for $i < -1$, if $x \in [2^i, 2^{i+1})$ then $\operatorname{erf} x \in [2^i, 2^{i+2})$. If $x \geq 0.5$, then $\operatorname{erf} x \in [0.5, 1)$. Thus we can apply the $\mathsf{FixToFloat}$ protocol on line 19 to the numbers $\llbracket\widetilde{g_i}\rrbracket$ to obtain floating point numbers $\llbracket N_{-1}\rrbracket, \dots, \llbracket N_w\rrbracket$ We additionally compute $\llbracket N_{-2}\rrbracket = \frac{2}{\sqrt{\pi}}\llbracket N\rrbracket$ and set $\llbracket N_{w+1}\rrbracket$ to $\llbracket 1\rrbracket$ on lines 21 and 22. In order to be able to use only the last $\log_2(2 + w)$ bits for the generalized oblivious choice, we set $\llbracket E\rrbracket$ to $q - w$ if it is smaller than $q - w$ and to $q + 3$ if it is larger than $q + 3$ on lines 24, 25, 27 and 28.

Then we use the generalised oblivious choice protocol on both exponents and significands on lines 29 and 30 to choose the final result between $\llbracket N_{-2}\rrbracket, \dots, \llbracket N_{w+1}\rrbracket$ based on the exponent $\llbracket E\rrbracket$. Note that if $x = 0$, then the oblivious choice will choose $\llbracket N_{-2}\rrbracket = \frac{2}{\sqrt{\pi}}\llbracket 0\rrbracket = \llbracket 0\rrbracket$ and thus the protocol is correct also when the input is zero.

Note that while it would have been possible to shift by a protected number of bits and thus obtain a single fixed-point number on which we could do polynomial evaluation, we still would have to use different polynomials for different value ranges of the input and perform oblivious choices between them and since shifting by a protected number of bits is an expensive operation we decided against it.

6 Results and Comparison

We have implemented four selected functions on the Sharemind 3 computing platform and benchmarked the implementations. To measure the performance of the floating point operations we deployed the developed software on three servers connected with fast network connections.

More specifically, each of the servers used contains two Intel X5670 2.93 GHz CPUs with 6 cores and 48 GB of memory. Since on Sharemind parallel composition of operations is more efficient than sequential composition, all the operations were implemented as vector operations. To see how much the vector size affects the average performance, we ran tests for different input sizes for all our inputs. We did 5 tests for each operation and input size and computed the average.

We compare here our results with previously existing protocols for computing the functions on either fixed-point values or floating-point values. How we reached the error estimates is described in the full version of the paper [13]. The error estimates are relative errors given for the significand, meaning that they will be respectively bigger or smaller when the exponent is bigger or smaller. As for previous work, Catrina and Dragulin achieve maximal possible precision given their used number of bits, Aliasgari *et al* achieve precision 2^{-32}, Liedel achieves precision 2^{-80} and Kamm and Willemson achieve precision that is approximately in the same orders of magnitude as this paper.

	1	10	100	1000	10000
Catrina, Dragulin, 128 bits, AppDiv2m, LAN(ms) [5]	3.39				
Catrina, Dragulin, 128 bits, Div2m, LAN(ms) [5]	1.26				
Kamm and Willemson, Chebyshev, 32 bits [11]	0.17	1.7	15.3	55.2	66.4
Kamm and Willemson, Chebyshev, 64 bits [11]	0.16	1.5	11.1	29.5	47.2
Current paper, 32 bits	0.99	8.22	89.73	400.51	400.51
Current paper, 64 bits	0.82	8.08	62.17	130.35	130.35

Table 1. Operations per second for different implementation of the inverse function for different batch sizes.

Table 1 compares previous results for computing the inverse with our results. Our results are up to 6 times faster than the previously existing implementations. We estimate the error to be no larger than $1.3 \cdot 10^{-4}$ for the 32 bit case and $1.3 \cdot 10^{-8}$ for the 64 bit case. We had $m = 25$ in the 32 bit case and $m = 52$ in the 64 bit case. In the 32 bit case the polynomial has rank 5 and in the 64 bit case it had rank 10.

	1	10	100	1000	10000
Liedel [15]	0.204				
Kamm and Willemson 32 bits [11]	0.09	0.85	7	24	32
Kamm and Willemson 64 bits [11]	0.08	0.76	4.6	9.7	10.4
Current paper, 32 bits	0.77	7.55	70.7	439.17	580.81
Current paper, 64 bits	0.65	6.32	41.75	78.25	119.99

Table 2. Operations per second for different implementation of the square root function for different input sizes.

Table 2 compares previous results for computing the square root with our results. Our results are up to 18 times faster than the best previously existing implementations. We estimate the error to be no larger than $5.1 \cdot 10^{-6}$ for 32 bit case and $4.1 \cdot 10^{-11}$ for the 64 bit case. We had $m = 31$ in the 32 bit case and $m = 52$ in the 64 bit case. In the 32 bit case the polynomial has rank 6 and in the 64 bit case it had rank 16.

	1	10	100	1000	10000
Aliasgari *et al.* [1]		6.3	9.7	10.3	10.3
Kamm and Willemson, (Chebyshev) 32 bits [11]	0.11	1.2	11	71	114
Kamm and Willemson, (Chebyshev) 64 bits [11]	0.11	1.1	9.7	42	50
Current paper, 32 bits	0.24	2.41	24.03	104.55	126.42
Current paper, 64 bits	0.23	2.27	16.66	47.56	44.84

Table 3. Operations per second for different implementation of the exponential function for different input sizes.

Table 3 compares previous results for computing the exponent with our results. Our results are up to 2 times faster than the best previously existing implementations. We estimate the error to be no larger than $6 \cdot 10^{-6}$ for 32 bit case and $1.5 \cdot 10^{-12}$ for the 64 bit case. We had $m = 30$ in the 32 bit case and $m = 62$ in the 64 bit case. In the 32 bit case the polynomial has rank 4 and in the 64 bit case it had rank 8.

Table 4 compares previous results for computing the Gaussian error function with our results. Our results are up to 4 times faster than the

	1	10	100	1000	10000
Kamm and Willemson, 32 bits [11]	0.1	0.97	8.4	30	39
Kamm and Willemson, 64 bits [11]	0.09	0.89	5.8	16	21
Current paper, 32-bit	0.5	4.41	30.65	45.42	49.88
Current paper, 64-bit	0.46	4.13	21.97	19.54	26.11

Table 4. Operations per second for different implementation of the Gaussian error function for different input sizes.

previously existing implementations. We estimate error for 32 bit case to be no greater than $1.1 \cdot 10^{-6}$ for inputs from $[0, 1)$, no greater than $7 \cdot 10^{-6}$ for inputs from $[1, 2)$, no greater than $1.5 \cdot 10^{-5}$ for inputs from $[2, 3)$ and no greater than $4 \cdot 10^{-6}$ for inputs from $[3, 4)$. We estimate error for 64 bit case to be no greater than $2 \cdot 10^{-8}$ in $[0, 1)$, no greater than $4 \cdot 10^{-9}$ in $[1, 2)$, no greater than 10^{-8} in $[2, 3)$ and no greater than $1 \cdot 10^{-7}$ in $[3, 4)$. We had $m = 26$ in the 32 bit case and $m = 51$ in the 64 bit case and $w = 4$. All the polynomials had rank 12.

7 Conclusion

We developed fixed-point numbers for the Sharemind secure multiparty computation platform. We improved on existing algorithms by [11] for floating-point numbers for the inverse, square-root, exponential and error functions by constructing a hybrid model of fixed-point and floating-point numbers. These new algorithms allow for considerably faster implementations than the previous ones.

References

1. Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS*, 2013.
2. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In Sushil Jajodia and Javier Lopez, editors, *ESORICS'08*, volume 5283 of *LNCS*, pages 192–206. Springer Berlin / Heidelberg, 2008.
3. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.
4. Dan Boneh, Giovanni Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In Christian Cachin and JanL. Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer Berlin Heidelberg, 2004.

5. Octavian Catrina and Claudiu Dragulin. Multiparty computation of fixed-point multiplication and reciprocal. In *Database and Expert Systems Application, 2009. DEXA '09. 20th International Workshop on*, pages 107–111, 2009.
6. Octavian Catrina and Sebastiaan Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security – ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 134–150. Springer Berlin Heidelberg, 2010.
7. Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *Financial Cryptography and Data Security*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer Berlin Heidelberg, 2010.
8. Morten Dahl, Chao Ning, and Tomas Toft. On secure two-party integer division. In AngelosD. Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 164–178. Springer Berlin Heidelberg, 2012.
9. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09*, pages 169–178, 2009.
10. Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer Berlin Heidelberg, 2011.
11. Liina Kamm and Jan Willemson. Secure floating-point arithmetic and private satellite collision analysis. Cryptology ePrint Archive, Report 2013/850, 2013. `http://eprint.iacr.org/`.
12. F. Kerschbaum, A. Schroepfer, A. Zilli, R. Pibernik, O. Catrina, S. de Hoogh, B. Schoenmakers, S. Cimato, and E. Damiani. Secure collaborative supply-chain management. *Computer*, 44(9):38–43, 2011.
13. Toomas Krips and Jan Willemson. Hybrid model of fixed and floating point numbers in secure multiparty computations. Cryptology ePrint Archive, Report 2014/221, 2014. `http://eprint.iacr.org/`.
14. Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *ISC '11*, volume 7001 of *LNCS*, pages 262–277, 2011.
15. Manuel Liedel. Secure distributed computation of the square root and applications. In MarkD. Ryan, Ben Smyth, and Guilin Wang, editors, *Information Security Practice and Experience*, volume 7232 of *Lecture Notes in Computer Science*, pages 277–288. Springer Berlin Heidelberg, 2012.
16. Y.-C. Liu, Y.-T. Chiang, T. s. Hsu, C.-J. Liau, and D.-W. Wang. Floating point arithmetic protocols for constructing secure data analysis application.
17. Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy Beyond K-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1), March 2007.
18. Pierangela Samarati. Protecting Respondents' Identities in Microdata Release. *IEEE Transactions on Knowledge and Data Engineering*, 13:1010–1027, 2001.
19. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
20. Latanya Sweeney. K-anonymity: A Model for Protecting Privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, October 2002.
21. Yihua Zhang, Aaron Steele, and Marina Blanton. Picco: A general-purpose compiler for private distributed computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 813–826, New York, NY, USA, 2013. ACM.