Secure Computation on Floating Point Numbers

Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele
Department of Computer Science and Engineering
University of Notre Dame
{maliasga,mblanton,yzhang16,asteele2}@nd.edu

Abstract

Secure computation undeniably received a lot of attention in the recent years, with the shift toward cloud computing offering a new incentive for secure computation and outsourcing. Surprisingly little attention, however, has been paid to computation with non-integer data types. To narrow this gap, in this work we develop efficient solutions for computation with real numbers in floating point representation, as well as more complex operations such as square root, logarithm, and exponentiation. Our techniques are information-theoretically secure, do not use expensive cryptographic techniques, and can be applied to a variety of settings. Our experimental results also show that the techniques exhibit rather fast performance and in some cases outperform operations on integers.

1 Introduction

As today's society is becoming increasingly connected with a growing amount of information available in the digital form, the amount of personal, sensitive, or otherwise private information collected and stored is constantly growing. When dealing with such information, one of the most technically challenging issues arises when private or sensitive information needs to be used in computation without revealing unnecessary information. This applies to both personal and proprietary data.

The desire to carry out computation in a privacy-preserving manner without revealing any information about the private inputs throughout the computation has been a topic of research since the first general secure function evaluation result was introduced by Yao in his seminal work [35]. Despite a large amount of attention from the research community, secure computation techniques are not commonly used in practice because of their complexity and overhead. The recent progress in the performance of secure multi-party computation techniques, however, shows that secure computation can be very fast (e.g., millions of operations performed on the order of seconds by Sharemind on a LAN [2]).

Furthermore, the shift toward cloud computing and storage offers a new major incentive for further development of such techniques. Cloud computing enables on-demand access to computing and data storage resources, which can be configured to meet unique constraints of the clients and utilized with minimal management overhead. Such services are attractive and economically sensible for clients with limited computing or storage resources who are unwilling or unable to procure and maintain their own computing infrastructure and can therefore outsource their computational tasks to the cloud. It has been suggested, however, that the top impediment on the way of harnessing the benefits of cloud computing to the fullest extent is security and privacy considerations that prevent clients from placing their data or computations on the cloud (see, e.g., [1]). Any progress in the performance of secure computation techniques or extending their scope toward *general-purpose computing over private data* can therefore make a substantial impact on the field of computing.

While it is well known that any computable function can be evaluated securely (e.g., as a Boolean or arithmetic circuit), recently a substantial amount of literature (see, e.g., [14, 15, 26] among many others) has been dedicated to optimizing secure realizations of commonly used operations. This is not surprising considering that, for instance, a substantial performance gain in secure implementation of a comparison operation (be it based on garbled circuits, homomorphic encryption, or secret sharing techniques) leads to a significant performance improvement of a very large number of functions which might be used in secure computation. To date, however, the great majority of techniques treat computation on the integers, while there are undeniably limitations to integer arithmetic. This work makes a step toward expanding the available techniques to other data types, and in particular develops techniques for secure multi-party computation on real numbers in the floating point representation. While prior literature [18, 12, 19] offers a limited number of techniques for real number computation, we are not aware of any work that develops a suite of techniques for floating point computation or secure multi-party techniques for complex functions such as logarithm, square root, and exponentiation. Such techniques are therefore the focus of this work.

Because performance of secure computation techniques is crucial, our techniques are designed to optimize both the overall and round complexity of the solutions. In addition, in order to provide a fair assessment of the performance of operations on different numeric data types in the multi-party framework, we implement a number of operations for integer, fixed point, and floating point real values and evaluate their performance. To the best of our knowledge, these are the first experimental results not only for floating point representation, but also for fixed point operations, which we carry out in a standard threshold linear secret sharing setting. To summarize, our contributions are as follows:

- Design of efficient secure multi-party techniques for floating point computation in a standard linear secret sharing framework.
- Design of efficient (and fast converging) protocols for complex operations over real numbers (square root, logarithm, and exponentiation).
- Evaluation of the developed and existing techniques for integer, fixed point, and floating point computation.

Security of our protocols is shown in both passive (also known as semi-honest or honest-but-curious) and active (malicious) adversarial models.

2 Preliminaries

2.1 Framework

In this work we use the multi-party setting in which $n > 2$ parties $P_1, \ldots, P_n$ jointly execute a prescribed functionality on private inputs and outputs. We utilize a linear secret sharing scheme (such as Shamir secret sharing scheme [33]) for representation of and secure computation on private values. To ensure composability of our protocols, we assume that prior to the computation, the parties P_1 through P_n hold their respective shares of the input and also compute shares of the output. Then any party holding a private input will produce shares of its values before the computation starts, and upon computation completion the computational parties P_1 through P_n send their shares to the entities that are entitled to learn the result. This gives flexibility to the problem setting in that the parties holding the inputs may be disjoint from the parties carrying out the computation (as in the case with outsourcing). Similarly, the parties receiving the output do not have to coincide with the input parties or computational parties.

Throughout this work we assume that parties $P_1, \ldots, P_n$ are connected by pair-wise secure authenticated channels. Each input and output party also establishes secure channels with P_1 through P_n. With a (n,t)-secret sharing scheme, any private value is secret-shared among n parties such that any $t+1$ shares can be used to reconstruct it, while t or fewer shares reveal no information about the shared value, i.e., it is perfectly protected in the information-theoretic sense. Therefore, the values of n and t should be chosen such that an adversary is unable to corrupt more than t computational parties.

In a secret sharing scheme that we utilize, any linear combination of secret-shared values can be performed by each computational party locally, without any interaction, but multiplication of two secret-shared values requires communication between all of them. In other words, if we let $[x]$ denote that value x is secret-shared among $P_1, \ldots, P_n$, operations $[x] + [y]$, $[x] + c$, and $c[x]$ are performed by each P_i locally on its shares of x and y, while computation of $[x][y]$ is interactive. We also use notation $\mathsf{Output}([a])$ to denote that all parties broadcast their shares of a which allows them to reconstruct the value of a. All operations are assumed to be performed in a field $\mathbb{F}_q$ for a small prime q greater than the maximum value that needs to be used in the computation (defined later).

Performance of secure computation techniques is of grand significance, as protecting secrecy of data throughout the computation often incurs substantial computational costs. For that reason, besides security, efficient performance of the developed techniques is one of our prime goals. Normally, performance of a protocol in the current setting is measured in terms of two parameters: (i) the number of interactive operations (multiplications, distributing shares of a private value or opening a secret-shared value) necessary to perform the computation, or *invocations*, and (ii) the number of sequential interactions, or *rounds*. We employ the same metrics throughout this work.

2.2 Related work

Prior work on general techniques for secure two- and multi-party computation is extensive, and its review is beyond the scope of this work. Prior work more closely related to ours primarily concentrates on techniques for integer computation and includes work on bit decomposition [14, 29, 34, 32], where a secret-shared value is converted to shares of the bits in its binary representation; comparison [14, 29, 21, 31], where the operands may or may not have to be given in the bit-decomposed form; addition and subtraction of bit decomposed values [14, 3, 10]; and division [4, 8, 11, 25, 12, 7]. The majority of the above techniques concentrate on constant-round protocols (i.e., independent of the bit length of their operands), and the researchers over time gained notable performance improvement for some of the operations.

Techniques for secure *two-party* computation of logarithm are also known [28], but they do not immediately generalize to the multi-party setting. Techniques for computing on real values are only starting to appear. In particular, Catrina and Saxena [12] use fixed point representation and develop techniques for several operations. Franz et al. [19] propose to use quantized logarithmic representation, which enables the relative error to be bounded for both very large and very small values, in the two-party setting (using homomorphic encryption and other tools). In addition, recently Franz et al. [20] proposed a framework for four basic operations on floating point numbers in the two-party setting using homomorphic encryption and garbled circuits with no implementation. Finally, Fouque et al. [18] provide two-party techniques for addition and multiplication using rational values. Given a sheer volume of applications that use real values and have been considered in prior literature in the secure computation context (e.g., scientific computing, biometric and signal processing, and data mining), additional work in standard frameworks is needed.

2.3 Building blocks

Our solutions rely on the following building blocks from prior literature:

- $[r] \leftarrow \mathsf{RandInt}(k)$ allows the parties to generate shares of a random k-bit value $[r]$ without any interaction (see, e.g., [12]) using what can be viewed as a distributed pseudo-random function.
- $[r] \leftarrow \mathsf{RandBit}()$ similarly allows the parties to produce shares of a random bit $[r]$ using one interactive operation.
- $[c] \leftarrow \mathsf{XOR}([a], [b])$ computes exclusive OR of bits a and b as $[a] + [b] - 2[a][b]$ using one multiplication.
- $[c] \leftarrow \mathsf{OR}([a], [b])$ computes OR of bits a and b as $[a] + [b] - [a][b]$.
- $[b] \leftarrow \mathsf{Inv}([a])$ computes $b = a^{-1}$ (in $\mathbb{F}_q$). For a non-zero a this operation can be implemented using a single interaction, where the parties create a random $[r]$, compute and open $c = r \cdot a$ and set $[b] = c^{-1}[r]$.
- $([y_1], \ldots, [y_n]) \leftarrow \mathsf{PreMul}([x_1], \ldots, [x_n])$ computes prefix-multiplication, where on input a sequence of integers $x_1, \ldots, x_n$, the output consists of values $y_1, \ldots, y_n$, where each $y_i = \prod_{j=1}^{i} x_j$. The most efficient implementation of this operation in our framework that we are aware of is due to Catrina and de Hoogh [10] that uses 2 rounds and $3n - 1$ interactive operations, but works only on non-zero elements of a field. Most of the cost of this protocol is input independent and can be performed ahead of time (after precomputation the cost becomes n invocations in 1 round).
- $([y_1], \ldots, [y_n]) \leftarrow \mathsf{PreOR}([x_1], \ldots, [x_n])$ computes prefix-OR of n input bits $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_n$ such that each $y_i = \bigvee_{j=1}^{i} x_j$. One possible implementation from [10] uses $5n-1$ invocations in 3 rounds, which after input-independent precomputation becomes $2n - 1$ invocations in 2 rounds.
- $[b] \leftarrow \mathsf{EQ}([x], [y], \ell)$ is an equality protocol that on input two secret-shared ℓ-bit values x and y outputs a bit b which is set to 1 iff $x = y$. Secure multi-party implementation of this operation in [10] uses another protocol $[b] \leftarrow \mathsf{EQZ}([x'], \ell)$, which outputs bit $b = 1$ iff $x' = 0$, by calling $\mathsf{EQZ}([x] - [y], \ell)$. EQ and EQZ thus both use $\ell + 4 \log \ell$ [1] invocation in 4 rounds, which after input-independent precomputation becomes $\log(\ell) + 2$ invocations in 3 rounds.
- $[b] \leftarrow \mathsf{LT}([x], [y], \ell)$ is a comparison protocol that on input two secret-shared ℓ-bit values x and y outputs a bit b which is set to 1 iff $x < y$. Efficient implementations of this function also exist, e.g., we can use the comparison protocol from [10] with 4 rounds and $4\ell - 2$ invocations (or $\ell + 1$ invocations in 3 rounds after precomputation). It similarly utilizes protocol $[b] \leftarrow \mathsf{LTZ}([x'], \ell)$ which outputs 1 iff $x' < 0$ by calling $\mathsf{LTZ}([x] - [y], \ell)$.
- $[y] \leftarrow \mathsf{Trunc}([x], \ell, m)$ computes $\lfloor [x]/2^m \rfloor$, where ℓ is the length of x. This operation can be efficiently implemented, e.g., using the techniques of [10], which require 4 rounds and $4m + 1$ invocations operations for this functionality, or $m + 2$ invocations in 3 rounds after input-independent precomputation.
- $[x_{m-1}], \ldots, [x_0] \leftarrow \mathsf{BitDec}([x], \ell, m)$ performs bit decomposition of m least significant bits of x, where ℓ is the size of x. An efficient implementation of this functionality can be found in [12] that uses $\log(m)$ rounds and $m \log(m)$ interactive operations (note that the complexity is independent of the size of its argument x).
- $[a] \leftarrow \mathsf{FPDiv}([x], [y], \gamma, f)$ performs division of floating point values x and y and produces floating point value x/y. It can be found in [12]. Here γ is the total length of floating point representation and 2^{-f} is the precision (i.e., there are $\gamma - f$ and f bits before and after the radix point, respectively). The complexity of this function for $\gamma = 2f$ is $3 \log(\gamma) + 2\theta + 12$

[1]Throughout this work, when we use $\log(a)$ we mean logarithm to the base 2, $\log_2(a)$.

rounds and $1.5\gamma\log(\gamma) + 2\gamma\theta + 10.5\gamma + 4\theta + 6$ interactive operations, where θ is the number of iterations equal to $\lceil\log(\gamma/3.5)\rceil$.

2.4 Security model

For each presented protocol, we define its secure functionality such that the parties carrying out the computation do not provide any input and do not receive any output. Instead, it is assumed that prior to the beginning of the computation the parties with inputs will secret-share their values among the parties carrying out the computation. Likewise, if the result of a computation is to be revealed to one or more parties, the computational parties will send their shares to the output parties who reconstruct the result. This allows for arbitrary composition of the protocols and their suitability for use in outsourced environments.

We next formally define security using the standard definition in secure multi-party computation for semi-honest adversaries. We prove our techniques secure in the semi-honest model and then show that standard techniques for making the computation robust to malicious behavior apply to our protocols as well.

Definition 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 $t < n$ and $\mathrm{VIEW}_\pi(I)$ 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). We say that protocol π is t-private in presence of semi-honest adversaries if for each coalition of size at most t there exists a probabilistic polynomial time simulator S_I such that

$$\{S_I(\mathsf{in}_I, f(\mathsf{in}_1, \ldots, \mathsf{in}_n))\} \equiv \{\mathrm{VIEW}_\pi(I), \mathsf{out}_I\},$$

where $\mathsf{in}_I = \bigcup_{P_i \in I}\{\mathsf{in}_i\}$, $\mathsf{out}_I = \bigcup_{P_i \in I}\{\mathsf{out}_i\}$, and $\equiv$ denotes computational indistinguishability.

3 New Building Blocks

Before proceeding with describing our solution for secure floating point arithmetic, we present new building blocks which are used in several of our protocols and can also be of independent interest. Such building blocks are:

- $[b] \leftarrow \mathsf{Trunc}([a], \ell, [m])$ that performs truncation of its first argument $[a]$ by an unknown number of bits m, where ℓ is the bitlength of a. This operation is the same as division by an unknown power of 2. Note that a straightforward approach to implementing this functionality is to run $\mathsf{Trunc}([a], \ell, m)$ on all possible values of $m < \ell$ and then obliviously choose one of them. This approach, however, results in $O(\ell^2)$ invocations, while we are able to achieve notable $O(\ell)$ in constant rounds. This, in particular, substantially outperforms general-purpose division. Our protocol is of independent interest and is used in this work in several types of operations on floating point numbers.

- $[a_0], \ldots, [a_{\ell-1}] \leftarrow \mathsf{B2U}([a], \ell)$ is a conversion procedure from a binary to unary representation. It converts the first argument $a < \ell$ from a binary to unary bitwise representation, where the output is ℓ bits, a least significant bits of which are set to 1 and all others are set to 0. In this work, B2U constitutes a significant component of our truncation protocol by an unknown number of bits, and its novelty and efficiency helps us to achieve the claimed performance of Trunc.

- $[2^a] \leftarrow \mathsf{Pow2}([a], \ell)$ raises 2 in the (unknown) power a supplied as the first argument to the function, where the second argument ℓ specifies the length of the values. In particular, the bitlength of 2^a cannot exceed ℓ, which means that a should be in the range $[0, \ell)$. Pow2 is called from many protocols including B2U, and is described next.

We implement $[2^a] \leftarrow \mathsf{Pow}([a], \ell)$ as follows: The logic behind it is rather simple and consists of first computing $m = \lceil \log \ell \rceil$ least significant bits $[a_{m-1}], \ldots, [a_0]$ of $[a]$ (since the result of raising 2 in the power of a longer than $\log(\ell)$-bit value cannot be represented) and computing the result as $\prod_{i=0}^{m-1}(2^{2^i}[a_i] + 1 - [a_i])$. This gives us complexity sub-linear in ℓ.

```
$[2^a] \leftarrow \mathsf{Pow2}([a], \ell)$

1. $m \leftarrow \lceil \log \ell \rceil$;
2. $[a_{m-1}], \ldots, [a_0] \leftarrow \mathsf{BitDec}([a], m, m)$;
3. $([x_0], \ldots, [x_{m-1}]) \leftarrow \mathsf{PreMul}(2^{2^0}[a_0] + 1 - [a_0], \ldots, 2^{2^{m-1}}[a_{m-1}] + 1 - [a_{m_1}])$;
4. return $[x_{m-1}]$;
```

In addition to using the above Pow2 protocol as a building block of Trunc and B2U, we utilize it to implement other functionalities in this work and it is thus of independent interest.

In the next protocol, binary-to-unary conversion procedure $\mathsf{B2U}([a], \ell)$ given below, we first call $\mathsf{Pow2}([a], \ell)$. We then create a random $(\ell + \kappa)$-bit value r, the ℓ least significant bits of which are secret shared by the parties, and broadcast $c = [2^a] + [r]$ (steps 2 and 3). Here κ is a statistical security parameter, which is also used in many building blocks listed in section 2.3. The function $Bits(c, \ell)$ in step 4 returns ℓ least significant bits of (public) c. Given the bits c_i's of 2^a blinded with a random value r, in step 5 we compute XOR of c_i and the corresponding bit r_i of r and store the resulting bits as x_i's. As a result of this operation, we obtain that because a least significant bits of 2^a are 0, $c_i = r_i$ and $x_i = 0$ for $i = 0, \ldots, a-1$. Also, because the ath bit of 2^a is 1, $c_a \neq r_a$ and $x_a = 1$. The remaining most significant bits x_i can contain arbitrary values because of carry bit propagation. What matters to us, however, is the position of the first (smallest numbered) non-zero value among the bits x_i. As the next step in the computation, we execute PreOR on the computed bits x_i's, as a result of which we obtain that a least significant bits $y_0, \ldots, y_{a-1}$ are 0, while the remaining $\ell - a$ bits $y_a, \ldots, y_{\ell-1}$ are all set to 1. This allows us to easily obtain bits of the unary representation of a by returning the complement of each computed bit.

```
$[a_0], \ldots, [a_{\ell-1}] \leftarrow \mathsf{B2U}([a], \ell)$

1. $[2^a] \leftarrow \mathsf{Pow2}([a], \ell)$;
2. for $i = 0$ to $\ell - 1$ in parallel $[r_i] \leftarrow \mathsf{RandBit}()$;
3. $c \leftarrow \mathsf{Output}([2^a] + 2^\ell \cdot \mathsf{RandInt}(\kappa) + \sum_{i=0}^{\ell-1} 2^i [r_i])$;
4. $(c_{\ell-1}, \ldots, c_0) \leftarrow Bits(c, \ell)$;
5. for $i = 0$ to $\ell - 1$ in parallel $[x_i] \leftarrow c_i + [r_i] - 2c_i[r_i]$;
6. $([y_0], \ldots, [y_{\ell-1}]) \leftarrow \mathsf{PreOR}([x_0], \ldots, [x_{\ell-1}])$;
7. for $i = 0$ to $\ell - 1$ in parallel $[a_i] \leftarrow 1 - [y_i]$;
```

```
8. return $[a_0], \ldots, [a_{\ell-1}]$;
```

Lastly, we describe truncation $\mathsf{Trunc}([a], \ell, [m])$ and provide its protocol below. In the solution, we first produce bit-wise unary representation of its argument $[m]$ using $\mathsf{B2U}$. Because $\mathsf{B2U}$ also computes $[2^m]$ at an intermediate step, we store that value as well instead of recomputing it. The computation starting from line 3 uses the structure of the regular truncation by a number of bits given in the clear as the starting point. In detail, we first choose ℓ random bits, form an m-bit random r' and an $(\kappa+\ell-m)$-bit random r'' (without knowledge of m in the clear), and broadcast the value of a protected with an $(\kappa+\ell)$-bit random $r = 2^m r'' + r'$ (lines 3–6). After performing modulo reduction of the result (using all possible powers of 2 as the moduli), we obtain $c'' = (a+r) \bmod 2^m$ (line 8). Because $a \bmod 2^m = c'' - r' + 2^m d$, where d is a bit which equals to 1 if $a \bmod 2^m + r' > 2^m$, we need to compute the value of d and compensate for the error. This is done on lines 9–10, where the returned result is $(a - (a \bmod 2^m))2^{-m}$. Note that the output of Trunc is meaningful only when $0 \leq m < \ell$.

```
[b] ← Trunc([a], ℓ, [m])
1. $\langle [x_0], \ldots, [x_{\ell-1}] \rangle, [2^m] \leftarrow \mathsf{B2U}([m], \ell)$;
2. $[2^{-m}] \leftarrow \mathsf{Inv}([2^m])$;
3. for $i = 0$ to $\ell - 1$ in parallel $[r_i] \leftarrow \mathsf{RandBit}()$;
4. $[r'] \leftarrow \sum_{i=0}^{\ell-1} 2^i [x_i][r_i]$;
5. $[r''] \leftarrow 2^\ell \cdot \mathsf{RandInt}(\kappa) + \sum_{i=0}^{\ell-1} 2^i (1 - [x_i])[r_i]$;
6. $c \leftarrow \mathsf{Output}([a] + [r''] + [r'])$;
7. for $i = 1$ to $\ell - 1$ in parallel $c'_i = c \bmod 2^i$;
8. $[c''] \leftarrow \sum_{i=1}^{\ell-1} c'_i([x_{i-1}] - [x_i])$;
9. $[d] \leftarrow \mathsf{LT}([c''], [r'], \ell)$;
10. $[b] \leftarrow ([a] - [c''] + [r'])[2^{-m}] - [d]$;
11. return $[b]$;
```

Note that because the protocol already computes $a \bmod 2^m$, we can use its steps to easily realize $\mathsf{Mod2m}([a], \ell, [m])$ function for computing modulo reduction for 2^m, where m is secret shared. To accomplish this, all that is needed is to replace line 10 in Trunc with $[b] \leftarrow [c''] - [r'] + [2^m][d]$.

The round complexity and number of invocations of the above protocols and other protocols for floating point computation that we develop are provided in Table 1.

4 Basic Operations in Floating Point Representation

4.1 Floating point representation

Today the floating point format is the most common representation for real numbers on computers due to its obvious precision advantages over fixed point or integer representation. Floating point representation consists of a fixed-precision significand v and an exponent p which specifies how the value is to be scaled (using a specified base), e.g., the value of the form $v \cdot 2^p$ when base is 2. To maximize precision and the number of representable values, floating point numbers are typically stored in normalized form, where the significand stores the leading non-zero bit and a fixed number of bits that follows. Then because 0 does not have such a representation, to preserve correctness of the computation, we choose to explicitly store with each value a bit which indicates whether the value is 0. For performance reasons we also store the sign bit separately; this prevents us from explicitly testing for 0 values and the sign during basic operations. We therefore represent each real value u as a 4-tuple (v, p, z, s) with base 2, where v is an ℓ-bit significand, p is a k-bit exponent, z is a bit which is set to 1 iff $u = 0$, and s is a sign bit which is set when the value is negative. We obtain that $u = (1 - 2s)(1 - z)v \cdot 2^p$. When $u = 0$, we also maintain that $z = 1$, $v = 0$, and $p = 0$.

Each non-zero value is normalized, which in our representation means that the most significant bit of v is always set to 1 and therefore $v \in [2^{\ell-1}, 2^{\ell})$. This is done by adjusting the (signed) exponent p accordingly, which is from the range $(-2^{k-1}, 2^{k-1})$ that we denote by $\mathbb{Z}_{\langle k \rangle}$. The mapping from negative values to elements of the field is straightforward: for any $a \in \mathbb{Z}_{\langle k \rangle}$, we define it as $\mathsf{fld}(a) = a \bmod q$. This representation is guaranteed to work correctly with both field operations (i.e., addition and multiplication), and other protocols (for instance, truncation) are also designed to work with this representation.

While error detection is not a common topic in secure multi-party literature, and error indicators can leak information about private data, for the type of operations developed in this work we consider error detection an important enough topic to address. This mechanism is optional, and if for certain computations the fact that an error occurred may result in a privacy leak, error detection can be avoided with the understanding that in the case of an error a valid, but meaningless result will be returned.

Among the five IEEE exceptions for floating point numbers, we only consider four main error types: *invalid operation* (e.g., square root or logarithm of a negative input), *division by zero* (e.g., division by 0 or logarithm of 0), *overflow*, and *underflow* (the absolute value of the output of an operation is either too large or too small to be represented using finite precision). The last IEEE exception for floating point numbers, inexact (the rounded result of a valid operation requires higher precision than what is used), is usually masked, and we therefore omit detection of such an error. Among all operations on floating point numbers that we consider in this work, the first two types of exceptions (i.e., invalid operation and division by zero) can only occur in division, square root, and logarithm. For such cases, we utilize a special flag, Error, which is set when one of these exceptions is triggered. For example, in the logarithm protocol, if the input is either 0 (division by zero exception) or negative (invalid operation exception), then the error flag is set. The other two types of exceptions, underflow and overflow, can occur in division, multiplication, addition, subtraction, and exponentiation operations. To check for these exceptions, one only needs to perform two comparisons at the end of protocols: If the exponent of the result is either greater than the maximum exponent value $2^{k-1}-1$ or it is less than the minimum exponent value $-2^{k-1}+1$, then overflow or underflow, respectively, has occurred. For simplicity of presentation, we do not explicitly include these two comparisons in our protocols. Furthermore, we leave them optional and do not include them in the implementation, as the probability of such errors occurring in practice with the adequate choice of parameters ℓ and k for floating point representation is very small.

As mentioned above, the error flag is set as part of operation execution. If it is not immediately revealed, the parties will need to maintain a global error indicator. This global error indicator is ORed with the error flag returned from an operation after each operation in which an exception can take place. The computational parties can handle errors in one of three ways as described in [20]. Namely, either the parties can open the error flag after each operation, the flag can be opened periodically after a certain number of operations, or it can be returned to the parties entitled to learn the result at the end of all of operations. What strategy is adopted is guided by the amount of information that the error flag may leak about the inputs. It is also trivial to extend our exception detection to a more fine-grained approach by using a dedicated flag for each type of exception.

In our protocols we require that $\mathbb{F}_q$ is such that integers of length $\max(2\ell + 1, k)$ can be represented (to accommodate expansion during multiplication or division). Also, if overflow and underflow detection is used, the above changes to $\max(2\ell + 1, k + 1)$.

4.2 Basic operations on floating point numbers

We next show how we can securely realize basic operations on real values in the floating point representation.

4.2.1 Multiplication and division

Multiplication of two floating point numbers $\langle v_1, p_1, z_1, s_1\rangle$ and $\langle v_2, p_2, z_2, s_2\rangle$ is performed by first multiplying their significands v_1 and v_2 to obtain a 2ℓ-bit product v. The product then needs to be truncated by either ℓ or $\ell - 1$ bits depending on whether the most significant bit of v is 1 or not. In the protocol below this is accomplished on lines 2–4 through oblivious execution. Partitioning truncation into two steps allows us to reduce the number of interactive operations (at a slight increase in the number of rounds). After computing the zero and sign bits of the product (lines 5 and 6), we obliviously adjust the exponent for non-zero values by the amount of previously performed truncation.

```
$\langle [v], [p], [z], [s]\rangle \leftarrow \mathsf{FLMul}(\langle [v_1], [p_1], [z_1], [s_1]\rangle, \langle [v_2], [p_2], [z_2], [s_2]\rangle)$

1. $[v] \leftarrow [v_1][v_2]$;
2. $[v] \leftarrow \mathsf{Trunc}([v], 2\ell, \ell - 1)$;
3. $[b] \leftarrow \mathsf{LT}([v], 2^\ell, \ell + 1)$;
4. $[v] \leftarrow \mathsf{Trunc}(2[b][v] + (1 - [b])[v], \ell + 1, 1)$;
5. $[z] \leftarrow \mathsf{OR}([z_1], [z_2])$;
6. $[s] \leftarrow \mathsf{XOR}([s_1], [s_2])$;
7. $[p] \leftarrow ([p_1] + [p_2] + \ell - [b])(1 - [z])$;
8. return $\langle [v], [p], [z], [s]\rangle$;
```

The complexity of this and other protocols in this section are given in Table 1. The numbers listed correspond to the minimal number of rounds and invocations, which often requires rearranging computation in the protocols. For example, computing $2[b][v] + (1 - [b])[v]$ on line 4 of FLMul uses only one multiplication; computation on lines 1, 5, and 6 of FLMul can be carried out in parallel, etc. When multiple possibilities exist for the building blocks, we use the complexities listed in section 2.3. Also, in certain computation we might need to multiply a floating point number by a constant. Then if one of the operands (or even only some components of one operand) is given in the clear, the complexity of this protocol reduces. In particular, multiplications on lines 1, 5, and 6 become local, which also reduces the number of rounds.

The main component of our division protocol is a simplified division SDiv given in Appendix A, which we modified from the fixed point division FPDiv in [12]. Because FPDiv uses Goldschmidt's method for division, it pays the price of normalizing the divisor in order to compute the initial approximation. With our floating point representation, however, all values are already normalized, and the initial approximation can be computed at very low cost. This is the first large difference between our implementation and FPDiv. Second, in order to avoid significant growth in the bitlength of intermediate values which takes place in FPDiv, we choose to split the representation of intermediate results. This allows us to avoid the increase in the size of the field $\mathbb{Z}_q$, but adds complexity to the algorithm. We provide additional details about SDiv in Appendix A.

As the first step of our floating point division protocol FLDiv, we execute simple division SDiv on the significands (line 1). Note that, in order to avoid executing it on a zero divisor, we ensure that the second argument to SDiv is non-zero. The output of SDiv is already normalized for our inputs and therefore no additional processing is needed. We then adjust the exponent p of the result. When z_2 is set we need to also set the error flag (division by zero if z_1 is not set and invalid operation when z_1 is also set).

Complexity of this protocol depends on value $\theta = \log(\ell/3.5)$.

```
$(\langle[v],[p],[z],[s]\rangle,[\mathsf{Error}]) \leftarrow \mathsf{FLDiv}(\langle[v_1],[p_1],[z_1],[s_1]\rangle,\langle[v_2],[p_2],[z_2],[s_2]\rangle)$

1. $[v] \leftarrow \mathsf{SDiv}([v_1],[v_2]+[z_2],\ell)$;
2. $[p] \leftarrow (1-[z_1])([p_1]-[p_2]-\ell)$;
3. $[z] \leftarrow [z_1]$;
4. $[s] \leftarrow \mathsf{XOR}([s_1],[s_2])$;
5. $[\mathsf{Error}] \leftarrow [z_2]$;
6. return $(\langle[v],[p],[z],[s]\rangle,[\mathsf{Error}])$;
```

4.2.2 Addition and subtraction

Addition and subtraction of floating point values is more involved due to the need to align the arguments to the same exponents. Also, we do not explicitly provide a subtraction protocol as subtraction $\mathsf{FLSub}(\langle[v_1],[p_1],[z_1],[s_1]\rangle,\langle[v_2],[p_2],[z_2],[s_2]\rangle)$ can be performed by calling $\mathsf{FLAdd}(\langle[v_1],[p_1],[z_1],$ $[s_1]\rangle,\langle[v_2],[p_2],[z_2],1-[s_2]\rangle)$.

In the addition protocol below, we first compute shares of the larger and the smaller of both the significands and exponents, denoted by v_{max}, v_{min}, p_{max}, and p_{min}, respectively (lines 1–7). Computation of the exponents is straightforward, while for significands, if the exponents p_1 and p_2 differ, the larger value v_{max} is set based on the summand with the larger exponent, otherwise, it is set based on the larger significand.

The logic used in the protocol depends on the signs of inputs (i.e., if they are the same, the values are being added; otherwise, they are being subtracted), and the difference s_3 (line 8) indicates which case takes place. We also need to look at the difference between the exponents, denoted by Δ. There are two cases to consider: (i) $\Delta > \ell$ and (ii) $0 \le \Delta \le \ell$. When $\Delta > \ell$ and $s_3 = 0$ (addition), there is no need to perform addition and the significand and exponent of the output are set to v_{max} and p_{max}, respectively.[2] When $\Delta > \ell$ and $s_3 = 1$ (subtraction), how the algorithm proceeds depends on whether $v_{max} > 2^{\ell-1}$ or otherwise $v_{max} = 2^{\ell}$. In the former case, to obtain the result, we need to subtract 1 from v_{max}, while setting the exponent to p_{max}. This is because when we subtract v_{min} from v_{max} the result will have the same exponent as p_{max} and its significand will be $v_{max}-1$ because we only keep ℓ most significant bits. In the latter case, $v_{max} = 2^{\ell-1}$, and the final result's significand will be $2^{\ell}-1$ bits long, from which ℓ most significant bits are all 1, and the exponent needs to be set to $p_{max}-1$. To accommodate both cases in one statement, we use $v_3 = 2(v_{max}-s_3)+1$ on line 11. If the former case occurs, then v_3 will have $\ell+1$ bits in which the least significant bit is extra and will be later removed. If the latter case occurs, v_3 will contain the correct ℓ-bit value, but the exponent will later need to be set by decrementing p_{max}.

When $\Delta \le \ell$, we keep the minimum exponent and shift the maximum significand by Δ bits. This is done by multiplying v_{max} by 2^{Δ}, where 2^{Δ} is obtained using $\mathsf{Pow2}$ function (line 10). Therefore $v_4 = v_{max}2^{\Delta} + (1-2s_3)v_{min}$ will have $\ell+\Delta\pm 1$ bits, which is at most 2ℓ. Next, we combine the two cases above (line 13) using the result of the comparison on line 9: v is either v_3 (when $\Delta > \ell$) or v_4 (when $\Delta \le \ell$). To make v of a fixed length we also multiply it by $2^{\ell-\Delta}$. Note that $\ell - \Delta$ is always positive. We obtain that now v will have $2\ell+\delta$ bits where δ is either 0 or ± 1. Because we can only keep ℓ most significant bits of v, we truncate $\ell-1$ least significant bits

[2] As with all other protocols, we store the most significant ℓ bits of the result by truncating all remaining bits when the intermediate results are longer than ℓ-bits long. This corresponds to rounding down the absolute value.

from the result to ensure that at least ℓ most significant bits of v are always kept(line 14). Now we need to normalize the result and remove the extra bits. To do so, we first bit-decompose v and using prefix-OR calculate the location p_0 of the most significant bit set to 1 in v's bit decomposition (lines 15–17). Then we shift v to left by p_0 bits by multiplying the value by 2^{p_0}. Now v has exactly $\ell + 2$ bits and it is normalized, from which two least significant bits are subsequently removed (line 18–19).

We also need to adjust the exponent of the result to reflect the shiftings and truncations performed. If $\Delta > \ell$, then $p_0 = 0$ when $v > 2^{\ell-1}$ and $p_0 = 1$ otherwise when $v = 2^{\ell-1}$. Thus the final exponent should be $p_{max} - p_0$. If $\Delta \leq \ell$, then since we truncate the result by $\Delta + 1$ and then we shift it by p_0, the original exponent p_{min} needs to be adjusted accordingly. This means that the final exponent p will be $p_{min} + \Delta + 1 - p_0 = p_{max} - p_0 + 1$ (line 20).

Lastly, we check whether any of the original summands are 0, in which case the previously computed sum can be incorrect, and the result instead is set to equal the non-zero summand (line 21).

The rest of the algorithm computes the remaining elements of the result. In particular, the zero bit is set based on whether the computed significand is 0. The exponent is adjusted similar to the adjustment of the significand on line 21 and takes into account zero values. The computation of the sign of the result on line 24 is similar to the computation of the larger significand on line 7 using the sign of the larger summand. The sign is then adjusted (similar to v and p) to take into account zero values (line 25).

```
$\langle [v], [p], [z], [s] \rangle \leftarrow \mathsf{FLAdd}(\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle)$

 1. $[a] \leftarrow \mathsf{LT}([p_1], [p_2], k);$
 2. $[b] \leftarrow \mathsf{EQ}([p_1], [p_2], k);$
 3. $[c] \leftarrow \mathsf{LT}([v_1], [v_2], \ell);$
 4. $[p_{max}] \leftarrow [a][p_2] + (1 - [a])[p_1];$
 5. $[p_{min}] \leftarrow (1 - [a])[p_2] + [a][p_1];$
 6. $[v_{max}] \leftarrow (1 - [b])([a][v_2] + (1 - [a])[v_1]) + [b]([c][v_2] + (1 - [c])[v_1]);$
 7. $[v_{min}] \leftarrow (1 - [b])([a][v_1] + (1 - [a])[v_2]) + [b]([c][v_1] + (1 - [c])[v_2]);$
 8. $[s_3] \leftarrow \mathsf{XOR}([s_1], [s_2]);$
 9. $[d] \leftarrow \mathsf{LT}(\ell, [p_{max}] - [p_{min}], k);$
10. $[2^{\Delta}] \leftarrow \mathsf{Pow2}((1 - [d])([p_{max}] - [p_{min}]), \ell + 1);$
11. $[v_3] \leftarrow 2([v_{max}] - [s_3]) + 1;$
12. $[v_4] \leftarrow [v_{max}][2^{\Delta}] + (1 - 2[s_3])[v_{min}];$
13. $[v] \leftarrow ([d][v_3] + (1 - [d])[v_4])2^{\ell}\mathsf{inv}([2^{\Delta}]);$
14. $[v] \leftarrow \mathsf{Trunc}([v], 2\ell + 1, \ell - 1);$
15. $[u_{\ell+1}], \ldots, [u_0] \leftarrow \mathsf{BitDec}([v], \ell + 2, \ell + 2);$
16. $[h_0], \ldots, [h_{\ell+1}] \leftarrow \mathsf{PreOR}([u_{\ell+1}], \ldots, [u_0]);$
17. $[p_0] \leftarrow \ell + 2 - \sum_{i=0}^{\ell+1} [h_i];$
18. $[2^{p_0}] \leftarrow 1 + \sum_{i=0}^{\ell+1} 2^i (1 - [h_i]);$
19. $[v] \leftarrow \mathsf{Trunc}([2^{p_0}][v], \ell + 2, 2);$
20. $[p] \leftarrow [p_{max}] - [p_0] + 1 - [d];$
21. $[v] \leftarrow (1 - [z_1])(1 - [z_2])[v] + [z_1][v_2] + [z_2][v_1];$
22. $[z] \leftarrow \mathsf{EQZ}([v], \ell);$
23. $[p] \leftarrow ((1 - [z_1])(1 - [z_2])[p] + [z_1][p_2] + [z_2][p_1])(1 - [z]);$
24. $[s] \leftarrow (1 - [b])([a][s_2] + (1 - [a])[s_1]) + [b]([c][s_2] + (1 - [c])[s_1]);$
25. $[s] \leftarrow (1 - [z_1])(1 - [z_2])[s] + (1 - [z_1])[z_2][s_1] + [z_1](1 - [z_2])[s_2];$
```

```
26. return $\langle [v], [p], [z], [s] \rangle$;
```

As in the case of FLMul, complexity of FLAdd can be reduced when one operand is a constant given in the clear. In this case, not only a number of multiplications can be performed locally, but also depending on the operand's value, some other operations might become unnecessary (e.g., the comparison on line 3).

Protocol	**Rounds**	**Invocations**
B2U	$\log\log\ell + 4$	$6\ell + 3\log\ell + (\log\ell)\log\log\ell - 1$
Trunc	$\log\log\ell + 9$	$12\ell + (\log\ell)\log\log\ell + 3\log\ell$
FLMul	11	$8\ell + 10$
FLDiv	$6\theta + 1$	$(8\theta - 1)\ell + 11\theta - 3$
FLAdd	$\log\ell + \log\log\ell + 27$	$14\ell + 9k + (\log\ell)\log\log\ell + (\ell + 9)\log\ell + 4\log k + 37$
FLLT	6	$9\ell + 4\log\ell + 7$
FLRound	$\log\log\ell + 30$	$15\ell + (\log\ell)\log\log\ell + 15\log\ell + 8k + 10$

Table 1: Complexity of floating point and supplemental protocols.

4.2.3 Additional operations

We conclude description of basic operations for the floating point representation by providing two other commonly use operations: comparisons and rounding.

The comparison operation below outputs bit b, which is set to 1 iff the first operand is less than the second operand. When neither of the operands is 0 and they have the same sign, the result can be determined by comparing the powers (and the values as well when the powers are equal). This computation appears on lines 1–4. Then lines 5–6 adjust the result to cover the cases when at least one operand is 0 or when the signs are different. Note that when both operands are 0, the output of this function is 0.

```
$[b] \leftarrow \mathsf{FLLT}(\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle)$

1. $[a] \leftarrow \mathsf{LT}([p_1], [p_2], k)$;
2. $[c] \leftarrow \mathsf{EQ}([p_1], [p_2], k)$;
3. $[d] \leftarrow \mathsf{LT}([v_1], [v_2], \ell)$;
4. $[b] \leftarrow [c][d] + (1 - [c])[a]$;
5. $[s] \leftarrow \mathsf{XOR}([s_1], [s_2])$;
6. $[b] \leftarrow [z_1](1 - [z_2])(1 - [s_2]) + (1 - [z_1])[z_2][s_1] + (1 - [z_1])(1 - [z_2])([s][s_1] + (1 - [s])[b])$;
7. return $[b]$;
```

Other relations (such as less than or equal, greater than, equality, etc.) can be easily constructed from the above protocol. It is not difficult to construct FLEQ protocol that avoids calls to LT and is more efficient.

The rounding function FLRound that we provide operates in two different modes: if mode $= 0$, it computes floor, and if mode $= 1$, it computes ceiling.

As the first step of our protocol (lines 1–2), we look at the input's exponent p_1. If $p_1 < -\ell + 1$, then the input value is from the range $(-1, 1)$ and thus the output should be either 0 or ± 1 depending on mode. In addition, if p_1 is positive, then the output of the rounding function is equal to the input (i.e., the input is an integer). The only case that requires non-trivial computation is when p_1 is between 0 and $-\ell$. In this case, rounding amounts to setting $-p_1$ least significant bits of the significand v_1 to 0. Furthermore, if those bits are not all 0, then based on the input's sign and mode we need to add 2^{-p_1} to the significand, the effect of which will be incrementing the integer value by 1. For instance, if we are to compute the floor of input -3.54, then the output should be $-(3+1) = -4$. This computation is performed on lines 3–5.

Now, if the addition on line 5 results in an overflow, then the result v needs to be adjusted. Because $2^{\ell-1} \le v_1 < 2^{\ell}$ and we are adding either 0 or 1 to it, the overflow happens only when $v = 2^{\ell}$. In this case, we set the value to $2^{\ell-1}$ (line 7) and increment the exponent by 1. To properly set the sign of the output, notice that the sign does not change unless the input belongs to the range $(-1, 0)$ and the function computes the ceiling. In those circumstances, we output is 0 and we to set the sign s to 0 as well (line 9). Lastly, we compute the zero bit of the result and adjust the exponent if necessary.

```
$\langle[v],[p],[z],[s]\rangle \leftarrow \mathsf{FLRound}(\langle[v_1],[p_1],[z_1],[s_1]\rangle, \mathsf{mode})$
  1. $[a] \leftarrow \mathsf{LTZ}([p_1], k)$;
  2. $[b] \leftarrow \mathsf{LT}([p_1], -\ell+1, k)$
  3. $\langle[v_2],[2^{-p_1}]\rangle \leftarrow \mathsf{Mod2m}([v_1], \ell, -[a](1-[b])[p_1])$;
  4. $[c] \leftarrow \mathsf{EQZ}([v_2], \ell)$;
  5. $[v] \leftarrow [v_1] - [v_2] + (1-[c])[2^{-p_1}](\mathsf{XOR}(\mathsf{mode}, [s_1]))$;
  6. $[d] \leftarrow \mathsf{EQ}([v], 2^{\ell}, \ell+1)$;
  7. $[v] \leftarrow [d]2^{\ell-1} + (1-[d])[v]$;
  8. $[v] \leftarrow [a]((1-[b])[v] + [b](\mathsf{mode} - [s_1])) + (1-[a])[v_1]$;
  9. $[s] \leftarrow (1-[b]\mathsf{mode})[s_1]$;
 10. $[z] \leftarrow \mathsf{OR}(\mathsf{EQZ}([v], \ell), [z_1])$;
 11. $[v] \leftarrow [v](1-[z])$;
 12. $[p] \leftarrow ([p_1] + [d][a](1-[b]))(1-[z])$;
 13. return $\langle[v],[p],[z],[s]\rangle$;
```

Above we use a slightly modified version of $\mathsf{Mod2m}([a], \ell, [m])$, which in addition to computing $[a \bmod 2^m]$ also outputs $[2^m]$, which is computed a part of the protocol. This allows us to avoid redundancy in computing $[2^{-p_1}]$.

We also note that our rounding functionality can be easily realized when mode is secret-shared. This allows the parties to execute input-based rounding operations. Furthermore, rounding to the nearest integer $\mathsf{FLRound}(\langle[v_1],[p_1],[z_1],[s_1]\rangle, \mathsf{mode} = 2)$ can be achieved by first adding 0.5 in the floating point representation to the input and then executing the floor function (i.e., $\mathsf{FLRound}(\mathsf{FLAdd}(\langle[2^{\ell-1}],[-\ell],0,0\rangle, \langle[v_1], [p_1],[z_1],[s_1]\rangle), 0)$).

5 Type Conversions

In this section we provide methods to convert a value from one representation (i.e., floating point, fixed point, or integer) to another. We use ℓ and k to denote bitlengths of significands and exponents, respectively, in a floating point representation. We also use γ to denote is the total bitlength of values in either integer or fixed point representation, and f to denote the bitlength of the fractional part in fixed point values (in other words, fixed point representation uses γ bits, from which f bits are dedicated to the fractional and $\gamma - f$ bits are dedicated to the integer part). Note that if x is represented as a fixed point value, then $x = \overline{x}2^{-f}$, where $\overline{x} \in \mathbb{Z}_{\langle\gamma\rangle}$. Thus representing $|\overline{x}|$ requires $\gamma - 1$ bits. Similarly, integer values $x \in \mathbb{Z}_{\langle\gamma\rangle}$ have $|x| < 2^{\gamma-1}$. Although the publicly known bitlengths ℓ, k, γ, f can be passed to protocols, we assume that $k > \max(\lceil\log(\ell+f)\rceil, \lceil\log(\gamma)\rceil)$ and the modulus $q > \max(2^{2\ell}, 2^{\gamma}, 2^{k})$.

Integer to floating point number. The first protocol that we provide converts a signed γ-bit integer a into a floating point value, and is given next. In the protocol, we first determine whether the input is 0 or is negative (lines 2 and 3 below) and turn a into a positive value on line 4. Because we need to normalize the value of a, we bit-decompose it into $\lambda = \gamma - 1$ bits (recall that one bit of its γ-bit representation was used for the sign). Prefix OR on line 6 allows us to compute the number of 0 bits in a starting from the most significant bit until the first 1 is observed. If the number of such bits is k, then our goal is to compute normalized value $2^k a$. We note that by using prefix OR, we set all bits after the most significant one. Thus if we flip all bits (i.e., 1-b_i) we have k ones and the rest of bits become zero. Now if we compose all the bits (i.e., $\sum 2^i(1-b_i)$) then we'll have $2^k - 1$. This computation and the final multiplication ($a2^k$) occurs on line 7. The output's exponent needs to be set to $-k$ (line 8) since a is now shifted k times to left.

What remains is to represent the value as an ℓ-bit number instead of using $\lambda = \gamma - 1$ bits and adjust the exponent accordingly. Therefore, if $\gamma - 1 > \ell$, we select ℓ most significant bits (line 9); otherwise, we shift the value by $\ell - \gamma + 1$ to the left to ensure that its most significant bit is 1 (line 10). The exponent is adjusted by the amount of shift on line 11 and reset to 0 if necessary.

```
$\langle [v], [p], [z], [s] \rangle \leftarrow \mathsf{Int2FL}([a], \gamma, \ell)$

1. $\lambda \leftarrow \gamma - 1$;
2. $[s] \leftarrow \mathsf{LTZ}([a], \gamma)$;
3. $[z] \leftarrow \mathsf{EQZ}([a], \gamma)$;
4. $[a] \leftarrow (1 - 2[s])[a]$;
5. $[a_{\lambda-1}], \ldots, [a_0] \leftarrow \mathsf{BitDec}([a], \lambda, \lambda)$;
6. $[b_0], \ldots, [b_{\lambda-1}] \leftarrow \mathsf{PreOR}([a_{\lambda-1}], \ldots, [a_0])$;
7. $[v] \leftarrow [a](1 + \sum_{i=0}^{\lambda-1} 2^i(1 - [b_i]))$;
8. $[p] \leftarrow -(\lambda - \sum_{i=0}^{\lambda-1} [b_i])$;
9. if $(\gamma - 1) > \ell$ then $[v] \leftarrow \mathsf{Trunc}([v], \gamma - 1, \gamma - \ell - 1)$;
10. else $[v] \leftarrow 2^{\ell-\gamma+1}[v]$;
11. $[p] \leftarrow ([p] + \gamma - 1 - \ell)(1 - [z])$;
12. return $\langle [v], [p], [z], [s] \rangle$;
```

Floating point number to integer. Our next protocol converts a floating point input with an ℓ-bit significand to the nearest integer representable in γ bits. As a first in our solution, we round the input to the nearest integer using FLRound with the result still being in the floating point format. The resulting integer can be represented using $\ell + p'$ bits (plus the sign), where p' is the exponent of the output, and our goal is to produce an γ-bit signed integer.

We divide all possible options into the following categories: If $p' \geq \gamma - 1$, the input is too large to be represented using γ bits. In other words, its last $p' \geq \gamma - 1$ bits are 0 and we output 0. This condition is checked on line 2 of our protocol, but setting the output to 0 is deferred until the end (this allows us not to worry about this case for most of the computation). Otherwise, if $p' + \ell > \gamma - 1$, the value is still too large to be represented using $\gamma - 1$ bits, and we have to truncate $\ell + p' - (\gamma - 1)$ most significant bits from the significand v'. In other words, the integer will consist of $\gamma - 1 - p'$ least significant bits of v' followed by p' 0 bits. This condition is checked on line 3. Now, if $p' < 0$, some of the significand's bits correspond to the fractional part (while being set to 0) and we need to truncate v' by p' bits. This condition is checked on line 4. When $p' < 0$, we, however, can still have that $p' + \ell > \gamma - 1$ (namely, when ℓ is much larger than γ), which means that a number of most significant bits of v' need to be truncated. Assuming that $p' < \gamma - 1$ we thus obtain four different cases based on conditions $p' > 0$ and $p' + \ell > \gamma - 1$.

The computation that follows treats the cases above. When $0 < p' < \gamma - 1$ and $p' + \ell > \gamma - 1$, we keep $\gamma - 1 - p'$ least significant bits of v' and otherwise keep v' unchanged (lines 5–7) (note that $1 - b - bc$ is the complement of $b(1 - c)$). The next 5 lines treat the cases when p' is negative. In particular, on lines 8–10, we remove p' least significant bits of v' (which are all 0) if p' is negative and keep v' unchanged otherwise. Also, $p' < 0$ and $p' + \ell > \gamma - 1$, we keep only $\gamma - 1$ least significant bits of the output (lines 11–12). What remains is to append p' 0 bits for positive p', include the sign, and set the value to 0 if necessary.

As an important performance issue, we note that two calls to Mod2m can be combined into a single call and, similarly, two calls to Pow2 in the protocol can be combined into one, thus noticeably improving the complexity of the algorithm. This is possible because either function is called on mutually exclusive arguments on two occasions (i.e., multiplied by bit c in one case and by bit $1 - c$ in the other), and the result can be updated as before. For readability of this work we do not include them in the protocol.

```
$[g] \leftarrow \mathsf{FL2Int}(\langle [v], [p], [z], [s] \rangle, \ell, k, \gamma)$

 1. $\langle [v'], [p'], [z'], [s'] \rangle \leftarrow \mathsf{FLRound}(\langle [v], [p], [z], [s] \rangle, 2)$;
 2. $[a] \leftarrow \mathsf{LT}([p'], \gamma - 1, k)$;
 3. $[b] \leftarrow \mathsf{LT}(\gamma - \ell - 1, [p'], k)$;
 4. $[c] \leftarrow \mathsf{LTZ}([p'], k)$;
 5. $[m] \leftarrow [a][b](1 - [c])(\gamma - 1 - [p'])$;
 6. $[u] \leftarrow \mathsf{Mod2m}([v'], \ell, [m])$;
 7. $[v'] \leftarrow [b](1 - [c])[u] + (1 - [b] + [b][c])[v']$;
 8. $[2^{-p'}] \leftarrow \mathsf{Pow2}(-[c][p'], \ell)$;
 9. $[2^{p'}] \leftarrow \mathsf{Inv}([2^{-p'}])$;
10. $[v'] \leftarrow ([c][2^{p'}] + 1 - [c])[v']$;
11. $[w] \leftarrow \mathsf{Mod2m}([v'], \ell, [b][c](\gamma - 1))$;
12. $[v'] \leftarrow [b][c][w] + (1 - [b][c])[v']$;
13. $[2^{p'}] \leftarrow \mathsf{Pow2}([a](1 - [c])[p'], \gamma - 1)$;
14. $[g] \leftarrow (1 - [z'])(1 - 2[s'])[2^{p'}][a][v']$;
15. return $[g]$;
```

Fixed point to floating point numbers. We next describe a protocol that converts a value $g \cdot 2^{-f}$ in fixed point notation to a floating point representation. We have that g is a signed γ-bit integer, and the resulting floating point number has a ℓ-bit significand and k-bit exponent. To perform the conversion, we call our integer-to-floating-point function Int2FL on the integer g and subtract f from the exponent of the output.

```
$\langle [v], [p], [z], [s] \rangle \leftarrow \mathsf{FP2FL}([g], \gamma, f, \ell, k)$

1. $\langle [v], [p], [z], [s] \rangle \leftarrow \mathsf{Int2FL}([g], \gamma, \ell)$;
2. $[p] \leftarrow ([p] - f)(1 - [z])$;
3. return $\langle [v], [p], [z], [s] \rangle$;
```

Floating point to fixed point numbers. Our last protocol converts a floating point number with a ℓ-bit significand and k-bit exponent to to a fixed point number represented using γ bits with precision 2^{-f}. This is achieved by converting the input to a γ-bit integer after adding the fixed point precision to the exponent. Note that if the input's exponent is close enough to the maximum so that if it overflows when f is added to it, the output should be set to zero.

```
$[g] \leftarrow \mathsf{FL2FP}(\langle [v],[p],[z],[s] \rangle, \ell, k, \gamma, f)$
  1. $[b] \leftarrow \mathsf{LT}([p], 2^{k-1} - f, k)$;
  2. $[g] \leftarrow \mathsf{FL2Int}(\langle [v],[p]+f,[z],[s] \rangle, \ell, k, \gamma)[b]$;
  3. return $[g]$;
```

6 Complex Operations

We next present several complex operations on numeric data types. While the approach we take to make the computation suitable for secure processing can apply to different data representations, the protocols that we provide focus on floating point values.

6.1 Square root

As the first operation, we would like to compute $x = \sqrt{y}$ for any positive real number y. To achieve this, one method is to use the old Babylonian formula that has very fast convergence rate:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{y}{x_i}\right)$$

where the initial point x_0 is always set to 1. The rate of convergence in this method is quadratic in the size of the input, which means that the number of bits of accuracy roughly doubles on each iteration. Then, if the precision is ℓ bits, the number of iterations is $\lceil \log \ell \rceil + 1$. As evident from the formula, this method requires one division, one addition, and one multiplication in each iteration. Because division is an expensive operation, there are other methods to approximate $\sqrt{x}$ using only multiplication and addition with the same rate of logarithmic convergence. One of these methods is Newton-Raphson's method. As described in [27], this method tries to iteratively approximate the roots of a continuous and at least one time differentiable function. We can apply Newton-Raphson's method to the function $f(R) = \frac{1}{R^2} - x$ to approximate $\frac{1}{\sqrt{x}}$ and ultimately approximate $\sqrt{x}$ by multiplying the approximation of $\frac{1}{\sqrt{x}}$ by x. The iterative equation in this method is

$$R_{i+1} = \frac{1}{2}R_i(3 - xR_i^2) \quad (1)$$

This method requires three multiplications and one addition inside the iterative equation. All these operations have to be executed sequentially thus the round complexity is high.

The most efficient (both in terms of round and invocation complexity) approach for computing the square root of a positive number is to use the iterative method of Goldschmidt [23]. It provides an approximation of both $\sqrt{x}$ and $\frac{1}{\sqrt{x}}$ with quadratic convergence. This method, as shown in [27], needs an initial approximation of $\frac{1}{\sqrt{x}}$ (denoted by y_0) that satisfies $0.5 < xy_0^2 < 1.5$. The value of y_0 is usually approximated using a linear equation $y_0 = \alpha x + \beta$, where α and β are predetermined constants that can be found in [27]. Then, if g is an approximation of $\sqrt{x}$ and h is an approximation of $\frac{1}{2\sqrt{x}}$, the iterative equations are:

$$g_{i+1} = g_i(1.5 - g_i h_i)$$
$$h_{i+1} = h_i(1.5 - g_i h_i)$$

where $h_0 = y_0/2$ and $g_0 = xy_0$. One of the advantages of this approach is that since g_{i+1} and h_{i+1} can be computed in parallel (independently) after $g_i h_i$ has been computed, the round complexity is reduced (as opposed to Newton-Raphson's method discussed above). This method is not self-correcting and the error can accumulate. To eliminate the accumulated errors, we take advantage of the self-correcting Newton-Raphson's approximation method and replace the last iteration of Goldschmidt's method with one iteration of Newton-Raphson's method. By replacing $R_i/2$ with h_i in equation (1), we obtain $h_{i+1} = h_i(3/2 - 2xh_i^2)$ as the new formula for approximating $\frac{1}{2\sqrt{x}}$. Ultimately, at the end of iterations, we need to multiply the last computed h_i by $2x$ to obtain an approximation for $\sqrt{x}$.

We next provide a protocol that implements Goldschmidt's approximation (similar to the fixed point protocol of [27]) for the floating point representation. In our representation, the significand of the input $2^{\ell-1} \le v < 2^\ell$ is already normalized and therefore $1/2 \le x = v2^{-\ell} < 1$. In fact, if we are to calculate the square root of $u = v \cdot 2^p$, we can run our approximation on $u = v2^{-\ell}2^{p+\ell} = x2^{p+\ell}$. Furthermore, $\sqrt{u} = \sqrt{ax}2^{\lfloor (p+\ell)/2 \rfloor}$, where a is either 1 (when $p+\ell$ is even) or 2 (when $p+\ell$ is odd). In the protocol, we first determine whether $p+\ell$ is even or odd (lines 1–2) and set the final exponent to $\lfloor (p+\ell)/2 \rfloor$ (line 3). Here ℓ_0 denotes the least significant bit of ℓ. Next, we compute the initial approximation y_0 using the linear equation mentioned above (lines 4–5). We use $\langle v_\alpha, p_\alpha, z_\alpha, s_\alpha \rangle$ and $\langle v_\beta, p_\beta, z_\beta, s_\beta \rangle$ to denote the floating point representation of α and β, respectively. The value of g_0 is computed on line 6 by multiplying x by the approximation of $\frac{1}{\sqrt{x}}$. To compute h_0, we only need to subtract 1 from the exponent of y_0 (line 7). In the loop (lines 8–12) we follow Goldschmidt's algorithm; but since our initial approximation has accuracy of 5.4 bits [27], we execute the loop $\lceil \log(\ell/5.4) \rceil - 1$ times followed by one iteration of Newton-Raphson's method (lines 13–17). Note that it is not necessary to compute line 11 in the last iteration of Goldschmidt's algorithm since g_i is not used after the loop. Finally, we need to multiply the result by the factor $\sqrt{a}$ above to obtain $\sqrt{v}$ (line 18). In the protocol, $v_{\sqrt{2}}$ is the ℓ-bit significand of $\sqrt{2}$ and $p_{\sqrt{2}}$ is its exponent in the floating point representation. Our floating point implementation of approximating square root has a substantially smaller complexity than the fixed point protocol of [27] for the same precision. This is due to the fact that in [27] quadratic invocation complexity is used to normalize the input while the input in our floating point representation is already normalized.

```
$(\langle [v], [p], [z], [s] \rangle, [\mathsf{Error}]) \leftarrow \mathsf{FLSqrt}(\langle [v_1], [p_1], [z_1], [s_1] \rangle)$

1. $[b] \leftarrow \mathsf{BitDec}([p_1], \ell, 1)$;
2. $[c] \leftarrow \mathsf{XOR}([b], \ell_0)$;
3. $[p] \leftarrow ([p_1] - [b])2^{-1} + \lfloor \ell/2 \rfloor + \mathsf{OR}([b], \ell_0)$;
4. $\langle [v_2], [p_2], [z_2], [s_2] \rangle \leftarrow \mathsf{FLMul}(\langle [v_1], -\ell, 0, 0 \rangle, \langle v_\alpha, p_\alpha, z_\alpha, s_\alpha \rangle)$;
5. $\langle [v_0], [p_0], [z_0], [s_0] \rangle \leftarrow \mathsf{FLAdd}(\langle [v_2], [p_2], [z_2], [s_2] \rangle, \langle v_\beta, p_\beta, z_\beta, s_\beta \rangle)$;
6. $\langle [v_g], [p_g], [z_g], [s_g] \rangle \leftarrow \mathsf{FLMul}(\langle [v_1], -\ell, 0, 0 \rangle, \langle [v_0], [p_0], [z_0], [s_0] \rangle)$;
7. $\langle [v_h], [p_h], [z_h], [s_h] \rangle \leftarrow \langle [v_0], [p_0] - 1, [z_0], [s_0] \rangle$;
8. for $i = 1$ to $\lceil \log(\ell/5.4) \rceil - 1$
9.     $\langle [v_2], [p_2], [z_2], [s_2] \rangle \leftarrow \mathsf{FLMul}(\langle [v_g], [p_g], [z_g], [s_g] \rangle, \langle [v_h], [p_h], [z_h], [s_h] \rangle)$;
10.    $\langle [v_2], [p_2], [z_2], [s_2] \rangle \leftarrow \mathsf{FLSub}(\langle 3 \cdot 2^{\ell-2}, -(\ell-1), 0, 0 \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle)$;
11.    $\langle [v_g], [p_g], [z_g], [s_g] \rangle \leftarrow \mathsf{FLMul}(\langle [v_g], [p_g], [z_g], [s_g] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle)$;
12.    $\langle [v_h], [p_h], [z_h], [s_h] \rangle \leftarrow \mathsf{FLMul}(\langle [v_h], [p_h], [z_h], [s_h] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle)$;
13. $\langle [v_{h^2}], [p_{h^2}], [z_{h^2}], [s_{h^2}] \rangle \leftarrow \mathsf{FLMul}(\langle [v_h], [p_h], [z_h], [s_h] \rangle, \langle [v_h], [p_h], [z_h], [s_h] \rangle)$;
14. $\langle [v_2], [p_2], [z_2], [s_2] \rangle \leftarrow \mathsf{FLMul}(\langle [v_1], -\ell, 0, 0 \rangle, \langle [v_{h^2}], [p_{h^2}], [z_{h^2}], [s_{h^2}] \rangle)$;
15. $\langle [v_2], [p_2], [z_2], [s_2] \rangle \leftarrow \mathsf{FLSub}(\langle 3 \cdot 2^{\ell-2}, -(\ell-1), 0, 0 \rangle, \langle [v_2], [p_2] + 1, [z_2], [s_2] \rangle)$;
16. $\langle [v_h], [p_h], [z_h], [s_h] \rangle \leftarrow \mathsf{FLMul}(\langle [v_h], [p_h], [z_h], [s_h] \rangle, \langle [v_2], [p_2], [z_2], [s_2] \rangle)$;
17. $\langle [v_2], [p_2], [z_2], [s_2] \rangle \leftarrow \mathsf{FLMul}(\langle [v_1], -\ell, 0, 0 \rangle, \langle [v_h], [p_h] + 1, [z_h], [s_h] \rangle)$;
18. $\langle [v_2], [p_2], [z_2], [s_2] \rangle \leftarrow \mathsf{FLMul}(\langle [v_2], [p_2], [z_2], [s_2] \rangle, \langle (1 - [c])2^{\ell-1} + [c]v_{\sqrt{2}}, -(1 - [c])(\ell - 1) +$
    $[c]p_{\sqrt{2}}, 0, 0\rangle)$;
19. $[p] \leftarrow ([p_2] + [p])(1 - [z_1])$;
20. $[v] \leftarrow [v_2](1 - [z_1])$;
21. $[\mathsf{Error}] \leftarrow [s_1]$;
22. return $(\langle [v], [p], [z_1], [s_1] \rangle, [\mathsf{Error}])$;
```

6.2 Exponentiation

We next treat exponentiation for the case that when the base is 2. Recall that exponentiation with an arbitrary base b can be computed as $b^a = 2^{a \log b}$, and we subsequently provide a solution for computing $\log b$ when b is a floating point number.

As any real number has an integer and fractional parts (e.g., 11001.01011), when 2 is raised to the power of that number, we break it into two components: 2 to the power of the integer part (in this example 2^{11001}) times 2 to the power of fractional part ($2^{0.01011}$). In fact, if we have the fractional part in a bit decomposed form $u_1, \cdots, u_n$, we can compute

$$2^{0.u_1 \cdots u_n} = \prod_{i=1}^{n} 2^{u_i 2^{-i}} = \prod_{i=1}^{n} (u_i 2^{2^{-i}} + (1 - u_i)) \tag{2}$$

For our floating point format, the smallest number of the form of $2^{2^{-i}}$ that we can represent is for $i = \ell - 1$. We therefore use a set of floating point constants $2^{2^{-i}}$ for $i = 1, \ldots, \ell - 1$. The above intuition leads to $\mathsf{FLExp2}$ protocol for computing exponentiation on a secret-shared floating point exponent that we describe next.

Given a floating point input, we would like to represent it as $w.u$, where w and u are the integer and fractional parts, respectively. For that reason, in $\mathsf{FLExp2}$ we first compute the integer part w by calling $\mathsf{FLRound}$ in the floor mode; this is to ensure that the fractional part u is always positive. Because of normalization of floating point values, if the exponent in floating point representation of w is negative, we need to truncate the extra 0's introduced by the normalization, which is done on line 3. Then if, on the other hand, the exponent is positive, the value of w is shifted by the appropriate number of bits to the left. The resulting w is then set as the initial exponent of the value $2^{w.u}$. In other words, p after the computation on line 5 represents 2^w.

The computation on line 6 produces $0.u$ in the floating point notation, the $\ell - 1$ bits of which after the radix point are turned into another floating point value on line 7 (this is accomplished by increasing the exponent of $0.u$ by $\ell - 1$ and computing the floor of the result). To convert the resulting floating point value into an integer, all is needed is to truncate the extra bits introduced by the normalization (note that this time the exponent p_u cannot be positive). This gives us integer u of size at most $\ell - 1$ bits, which is decomposed into bits on line 9.

The loop on lines 10–12 computes the terms in the product in equation 2. Here we use constants $\langle cv_i, cp_i, 0, 0 \rangle$ to represent $2^{2^{-i}}$ in the floating point format, and thus a_i's and b_i's correspond to the significand and exponent of $(u_i 2^{2^{-i}} + (1 - u_i))$, respectively. What remains is to multiply all of these terms, after which they need to be multiplied by 2^w, which is accomplished on lines 13–14. Note that the latter is accomplished by representing 2^w in the normalized form as $v \cdot 2^p = 2^{\ell-1} \cdot 2^{w-(\ell-1)}$.

The notation FLProd denotes the computation of the product of a number of floating point values. Instead of a number of $\ell - 2$ sequential multiplications in our case, it can be implemented in $\lceil \log(\ell - 1) \rceil$ rounds by constructing a tree of pairwise multiplications (i.e., $(\ell - 1)/2$ multiplications at the lowest level, $(\ell - 1)/4$ multiplications at the next level, etc.). With the availability of constant round prefix multiplication for floating point numbers, the round complexity of this operation could be reduced to constant rounds. We leave this as a direction for future work.

$\langle [v], [p], [z], [s] \rangle \leftarrow \mathsf{FLExp2}(\langle [v_1], [p_1], [z_1], [s_1] \rangle)$

```
1.  $\langle [v_w], [p_w], [z_w], [s_w] \rangle \leftarrow \mathsf{FLRound}(\langle [v_1], [p_1], [z_1], [s_1], 0 \rangle)$;
2.  $[b] \leftarrow \mathsf{LTZ}([p_w], k)$;
3.  $[v_w] \leftarrow \mathsf{Trunc}([v_w], \ell, -[b][p_w])[b] + (1 - [b])[v_w]$;
4.  $[2^{p_w}] \leftarrow \mathsf{Pow2}((1 - [b])[p_w], k - 1)$;
5.  $[p] \leftarrow (1 - 2[s_w])[2^{p_w}][v_w]$;
6.  $\langle [v_u], [p_u], [z_u], [s_u] \rangle \leftarrow \mathsf{FLSub}(\langle [v_1], [p_1], [z_1], [s_1] \rangle, \langle [v_w], [p_w], [z_w], [s_w] \rangle)$;
7.  $\langle [v_u], [p_u], [z_u], [s_u] \rangle \leftarrow \mathsf{FLRound}(\langle [v_u], [p_u] + \ell - 1, [z_u], [s_u] \rangle, 0)$;
8.  $[u] \leftarrow \mathsf{Trunc}([v_y], \ell, -[p_y])$;
9.  $[u_{\ell-1}], \ldots, [u_1] \leftarrow \mathsf{BitDec}([u], \ell, \ell - 1)$;
10. for $i = 1$ to $\ell - 1$ do in parallel
11.     $[a_i] \leftarrow 2^{\ell-1}(1 - [u_i]) + (cv_i)[u_i]$;
12.     $[b_i] \leftarrow -(\ell - 1)(1 - [u_i]) + (cp_i)[u_i]$;
13. $\langle [v_u], [p_u], 0, 0 \rangle \leftarrow \mathsf{FLProd}(\langle [a_1], [b_1], 0, 0 \rangle, \ldots, \langle [a_{\ell-1}], [b_{\ell-1}], 0, 0 \rangle)$;
14. $\langle [v], [p], [z], [s] \rangle \leftarrow \mathsf{FLMul}(\langle 2^{\ell-1}, [p] - (\ell - 1), 0, 0 \rangle, \langle [v_u], [p_u], 0, 0 \rangle)$;
15. return $\langle [v], [p], [z], [s] \rangle$;
```

6.3 Logarithm

The last operation that we present is computing logarithm to the base 2. Recall that for other bases b the function can be computed as $\log_b(x) = \log_2(x)/\log_2(b)$. In the following description, we use e to represent the Euler's constant.

To compute the logarithm of a positive number in any representation, we can utilize of the formula

$$\log x = 2 \log e \arctan \frac{x-1}{x+1}$$

and approximate $\arctan x$ by bounding its Taylor's series as $\arctan x \approx \sum_{i=0}^{M} \frac{x^{2i+1}}{(2i+1)}$ for $|x| < 1$. This gives us

$$\log x \approx 2 \log e \sum_{i=0}^{M} \frac{z^{2i+1}}{(2i+1)},$$

where $z = \frac{x-1}{x+1}$. In general, we have that $z \le z_{max} = \frac{x_{max}-1}{x_{max}+1}$, where x_{max} is the maximum value of input x, and to achieve ℓ-bit precision, we need to use $M > \lceil \ell/(-2 \log z_{max}) - 1/2 \rceil$. For practical values of ℓ, this approach finishes faster than using Taylor's series, since not only the latter approach requires more interactive rounds of computation but also finding the initial approximation is not trivial.

In our floating point representation, a positive value is represented as $x = v \cdot 2^p$, where $2^{\ell-1} \le v < 2^{\ell}$. We can therefore write v as $2^{\ell} u$ where $1/2 \le u < 1 - 2^{-\ell}$. In this representation, $\log x = \ell + p + \log u$, and we use the above approximation to derive the formula:

$$\log x = \ell + p - \sum_{i=0}^{M} 2 \log e \frac{y^{2i+1}}{2i+1} \tag{3}$$

where $y = (1 - v2^{-\ell})/(1 + v2^{-\ell})$. This computation requires one division, $2(M+1)$ multiplications, and $M+2$ additions, where $M = \lceil \ell/(2 \log 3) - 1/2 \rceil$ to achieve ℓ bits of precision. Note that one can apply Padé approximants [5] to the Taylor's series above to reduce the round complexity at the cost of (noticeably) increasing the invocation complexity of the protocol. We omit the details of such a technique.

Throughout most of our protocol FLLog2 below for computing log of floating point values, we assume that the input is positive. If the input is either 0 (the log of which is $-\infty$, division by zero exception) or negative (the log of which is a complex number, invalid operation exception), the error flag will be set (line 16).

The first 3 lines of the protocol compute $y = (1 - v_1 2^{-\ell})/(1 + v_1 2^{-\ell})$ in the floating point notation, followed by the computation of y^2 on line 4. Because the powers of y in the series above are all odd, in each iteration, we multiply the previous $y^{2(i-1)+1}$ by y^2 to obtain y^{2i+1} (line 9). In the protocol, we use constants $\langle cv_i, cp_i, 0, 0\rangle$ for $i = 0, \ldots, M$ to denote the floating point representation of $\frac{2 \log e}{(2i+1)}$. The variable (v, p) holds the current value of the sum in equation 3 (initialized on line 5 to the 0th constant times y) and the variable (v_y, p_y) holds the current power of y. (The variables with subscripts 2 and 3 hold intermediate results.) Then inside the loop we update the current power of y, multiply it by the appropriate constant, and add the result to the sum (lines 7–9).

After exiting the loop, we need to subtract the computed sum from $p + \ell$. To do so, we first convert $p + \ell$ into the floating point format (line 10) and then perform the subtraction (line 11) to obtain the final result. The rest of the function sets the output to 0 in case the input was 1 (represented as $\langle 2^{\ell-1}, -(\ell - 1), 0, 0\rangle$) and performs error checking.

```
$(\langle[v], [p], [z], [s], \rangle[\mathsf{Error}]) \leftarrow \mathsf{FLLog2}(\langle[v_1], [p_1], [z_1], [s_1]\rangle)$

 1. $\langle[v_2], [p_2], 0, 0\rangle \leftarrow \mathsf{FLSub}(\langle 2^{\ell-1}, -(\ell-1), 0, 0\rangle, \langle[v_1], -\ell, 0, 0\rangle)$;
 2. $\langle[v_3], [p_3], 0, 0\rangle \leftarrow \mathsf{FLAdd}(\langle 2^{\ell-1}, -(\ell-1), 0, 0\rangle, \langle[v_1], -\ell, 0, 0\rangle)$;
 3. $(\langle[v_y], [p_y], 0, 0\rangle, 0) \leftarrow \mathsf{FLDiv}(\langle[v_2], [p_2], 0, 0\rangle, \langle[v_3], [p_3], 0, 0)$;
 4. $\langle[v_{y^2}], [p_{y^2}], 0, 0\rangle \leftarrow \mathsf{FLMul}(\langle[v_y], [p_y], 0, 0\rangle, \langle[v_y], [p_y], 0, 0\rangle)$;
 5. $\langle[v], [p], 0, 0\rangle \leftarrow \mathsf{FLMul}(\langle[v_y], [p_y], 0, 0\rangle, \langle cv_0, cp_0, 0, 0\rangle)$;
 6. for $i = 1$ to $M$ do in parallel
 7.     $\langle[v_y], [p_y], 0, 0\rangle \leftarrow \mathsf{FLMul}(\langle[v_y], [p_y], 0, 0\rangle, \langle[v_{y^2}], [p_{y^2}], 0, 0\rangle)$;
 8.     $\langle[v_2], [p_2], 0, 0\rangle \leftarrow \mathsf{FLMul}(\langle[v_y], [p_y], 0, 0\rangle, \langle cv_i, cp_i, 0, 0\rangle)$;
 9.     $\langle[v], [p], 0, 0\rangle \leftarrow \mathsf{FLAdd}(\langle[v], [p], 0, 0\rangle, \langle[v_2], [p_2], 0, 0\rangle)$;
10. $\langle[v_2], [p_2], [z_2], [s_2]\rangle \leftarrow \mathsf{Int2FL}(\ell - [p], \ell, \ell)$;
11. $\langle[v], [p], [z], [s]\rangle \leftarrow \mathsf{FLSub}(\langle[v_2], [p_2], [z_2], [s_2]\rangle, \langle[v], [p], 0, 0\rangle)$;
12. $[a] \leftarrow \mathsf{EQ}([p_1], -(\ell-1), k)$;
13. $[b] \leftarrow \mathsf{EQ}([v_1], 2^{\ell-1}, \ell)$;
14. $[z] \leftarrow [a][b]$;
15. $[v] \leftarrow [v](1 - [z])$;
16. $[\mathsf{Error}] \leftarrow \mathsf{OR}([z_1], [s_1])$;
17. $[p] \leftarrow [p](1 - [z])$;
18. return $(\langle[v], [p], [z], [s]\rangle, [\mathsf{Error}])$;
```

7 Security Analysis

Correctness of the computation has been discussed with each respective protocol, and we now proceed with showing their security.

First, we note that the linear secret sharing scheme achieves perfect secrecy in presence of collusions of size at most t (i.e., zero information can be learned about secret-shared values by t or fewer parties) in the case of passive adversaries. Similarly, the multiplication protocol does not reveal any information, as the only information transmitted to the participants are the shares. Furthermore, because other building blocks used in this work (e.g., EQ, PreMul, etc.) have been previously shown to be secure, information is not revealed during their execution as well. The only value that is revealed in all of our protocols is c in Trunc. It, however, statistically hides any sensitive information, and the parties will not be able to learn any information about private values with non-negligible probability. We obtain that our protocols combine only secure building blocks without revealing any information to the computational parties (i.e., they only receive shares which information-theoretically protect private values). By Cannetti's composition theorem [9], we obtain that composition of secure sub-protocols results in security of the overall solution. More formally, to comply with the security definition, we can build a simulator for our protocols by invoking simulators for the corresponding building blocks to result in the environment that will be indistinguishable from the real protocol execution by the participants.

To show security in presence of malicious adversaries, we need to ensure that (i) all participants prove that each step of their computation was performed correctly and that (ii) if some dishonest participants quit, others will be able to reconstruct their shares and proceed with the rest of the computation. The above is normally achieved using a verifiable secret sharing scheme (VSS), and a large number of results have been developed over the years (e.g., [22, 24, 6, 17, 16] and others). In particular, because any linear combination of shares is computed locally, each participant is required to prove that it performed each multiplication correctly on its shares. Such results normally work for $t < \frac{n}{3}$ in the information theoretic or computational setting with different communication overhead and under a variety of assumptions about the communication channels. Additional proofs associated with this setting include proofs that shares of a private value were distributed correctly among the participants (when the dealer is dishonest) and proofs of proper reconstruction of a value from its shares (when not already implied by other techniques). In addition, if at any point of the computation the participants are required to input values of a specific form, they would have to prove that the values they supplied are well formed. Such proofs are not necessary for the computation that we use to construct our protocols, but are needed by the implementations of some of the building blocks (e.g., RandInt).

Thus, security of our protocols in the malicious model can be achieved by using standard VSS techniques, e.g., [22, 13], where a range proof, e.g., [30] will be additionally needed for the building blocks. These VSS techniques would also work with malicious input parties (who distribute inputs among the computational parties), who would need to prove that they generate legitimate shares of their data.

8 Experimental Results

In order to compare performance of arithmetic using different data types, we implement functions for integer, fixed point, and floating point operations. In the experiments, for floating point values we use bitlengths $\ell = 32$ for significands and $k = 9$ for (signed) exponents. This allows us to represent real values with precision 2^{-256}. For (signed) fixed point representation, we use bitlength $\gamma = 64$ with $f = 32$ (i.e., 32 bits before and after the radix point). This gives us precision 2^{-32} while covering values with the integer part up to 31 bits. Lastly, for (signed) integer computation we use length $\gamma = 64$, which makes it easier to compare performance with that of fixed point values (recall that a fixed point value g is represented as an integer $g \cdot 2^f$).

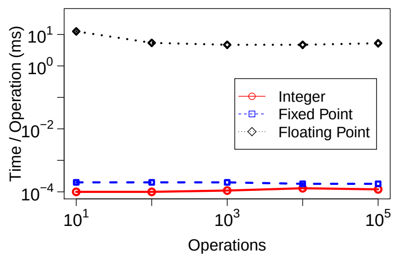
We set statistical hiding parameter $\kappa = 48$ for all data types. For integer arithmetic, the size of the field is determined by the bitlengths necessary for division and we obtain $|q| > 2\gamma + \kappa + 1 = 177$ (if division is known to not be used, the field size can be reduced to use $|q| > \gamma + \kappa = 112$). For fixed point arithmetic, the field size is also determined by the length of the values used during division and we obtain $|q| > \gamma + 3f + \kappa = 208$ (once again, if division is not used, the size of the field can be reduced to use $|q| > \gamma + f + \kappa = 144$). Lastly, for floating point arithmetic, we need to have $|q| > 2\ell + \kappa + 1 = 113$.

For our experiments, all protocols were implemented in C/C++. The computational parties were run on 2.4 GHz Linux machines on a 1Gbps LAN. We used (3, 1)-Shamir secret sharing with $n = 3$ participants. The floating point operations were implemented using protocols FLAdd, FLMul, FLLT, and FLDiv. Integer and fixed addition are the same as addition in $\mathbb{F}_q$. Multiplication of integer values is the same as multiplication in $\mathbb{F}_q$, while fixed point multiplication uses FPMul from [12]. Comparison for both integers and fixed point values is implemented using LT. Finally, fixed point division uses FPDiv from [12], while our implementation of integer division uses SDiv, where the initial approximation is computed in the same way as in FPDiv. The use of SDiv for integer division prevents the size of modulus from growing to $4\gamma + \kappa$.

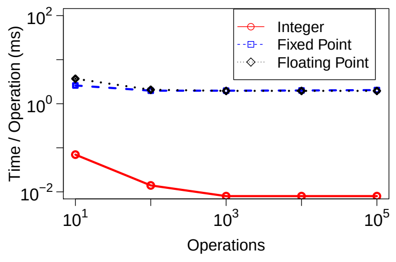
Our current implementation supports only a limited degree of parallel execution. In particular, we execute a number of identical operations in a single batch saving on communication and round complexity, but no support for fully parallelized execution is implemented. This in particular means that two different types of operations which could be executed independently in a protocol have to run sequentially in our implementation. In addition, although initial rounds and a number of interactive operations in some building blocks are input-independent and could be executed in parallel with the computation that precedes them, in our implementation they are run sequentially. This means that our implementation does not achieve the round complexity reported in Table 1 and the runtime of our protocols can be improved, especially when a small number of instances of an operation needs to be executed. This also implies that computational overhead of our implementation could be improved by utilizing multiple cores in an implementation with full support for parallelism.

While we attempted to optimize the implementation, there are a number of optimizations which, if implemented, would likely improve the performance of the protocols. They include using a smaller field size (e.g., $\mathbb{F}_{2^8}$) for computation on bits, which are subsequently converted to elements of $\mathbb{F}_q$ for the remaining computation. This is expected to make a larger performance difference for large moduli q since the conversion process adds overhead. We also note that for some of our floating point protocols (namely, FLAdd and FLLT) combined execution of comparisons which are run on exactly the same input (e.g., LT and EQ executed on $[p_1], [p_2]$) can be optimized by generating fewer random bits compared to when these algorithms are run separately. This is due to the fact that these algorithms first generate a number of random values, broadcast the result of adding the random values to the input, and use the broadcasted value to proceed with the rest of the computation. When such functions are executed on the same input, we can safely use a single broadcasted value for two different purposes.

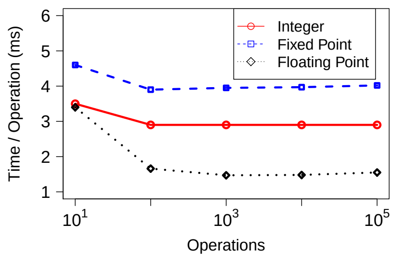
Figure 1 provides the results of the experiments for all three data types. The timing results are given per operations when a number of them were run in parallel. As the plots suggest, addition is very fast for both integer and fixed point data types as it is non-interactive, while for floating point values, the algorithm is quite involved and requires the values to be aligned before they can be added (note that with conventional, non-secure floating point computation, addition is more costly than multiplication). The difference in the performance of integer and fixed point addition is due to the different field sizes. The multiplication operation is substantially faster for integers than the two other data types, as it requires one invocation for integer values, while both fixed point and floating point multiplication involves truncation of the result. Comparison (less than), on the other hand, has similar performance for all three data types, with the floating point comparisons outperforming two other types. The small difference between integer and fixed point comparisons (which use



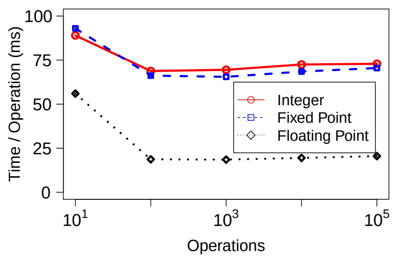
(a) addition



(b) multiplication



(c) comparison



(d) division

Figure 1: Performance of basic operations for different data types.

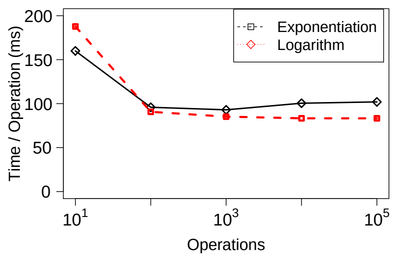


Figure 2: Performance of exponentiation and logarithm for floating point data type.

the same algorithm) can be explained by the difference in the field sizes, while faster performance of floating point comparisons is due to the need to compare shorter values despite using a more complex algorithm. Note that from all of these operations, each protocol takes at most a few msec. The remaining division operation takes the longest to execute with the floating point algorithm being almost an order of magnitude faster than the other two. This can be explained by the need of both integer and fixed point division to normalize the input for the initial approximation, which is not needed with (normalized) floating point representation. Note that in the attempt to avoid a larger growth of the field size for integer division, we implemented a solution with larger overhead, which shows in the figure that integer is slightly slower than fixed point division.

Figure 2 provides the timing results for exponentiation and logarithm of floating point numbers. The overhead of exponentiation is dominated FLProd operation (line 13 of FLExp2) that accounts for 2/3 of the overall time. Therefore, any direct improvement in the performance of FLProd will result in the same degree of improvement to FLExp2. We also observed that for a small number of operations (e.g., 10), the log-round implementation of FLProd results in significant savings in performance ($\approx$ 80%) compared to its sequential implementation, while for a large number of operations (e.g., 10,000), the improvement is barely noticeable ($\approx$ 3%). This complies with the fact that the round complexity is the bottleneck for a small number of parallel operations, while for a larger number of operations computation becomes the bottleneck.

9 Conclusions

This work introduces a number of secure and efficient multi-party protocols for floating point arithmetic including complex operations and conversion between different numeric data types. Our solutions are information-theoretically secure in a standard framework against passive and active adversaries. Implementation results also confirm suitability of the proposed techniques for a large variety of computations, and even show that secure computation over floating point numbers can in some cases outperform computation on integer or fixed point data types. While we treat a number of complex functions such as logarithm, exponentiation, and square, this work can be extended to other operations such as trigonometric functions. Lastly, additional optimizations and precise evaluation and bounding of errors introduced by the algorithms are also venues for future work.

References

[1] IT cloud services user survey, pt. 2: Top benefits & challenges. http://blogs.idc.com/ie/?p=210.

[2] Sharemind 2.0 reaches a million operations per second, December 2010. http://sharemind.cyber.ee/news-blog/sharemind-20-reaches-a-million-operations-per-second.

[3] SecureSCM Project Deliverable D9.2. http://pi1.informatik.uni-mannheim.de/index.php?pagecontent=site/Research.menu/SecureSCM.page, University of Mannheim, July 2009.

[4] M. Atallah, M. Bykova, J. Li, K. Frikken, and M. Topkara. Private collaborative forecasting and benchmarking. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 103–114, 2004.

[5] George A Baker and P. R. Graves-Morris. *Padé approximants.* Cambridge University Press, 1995.

[6] Z. Beerliova-Trubiniova and M. Hirt. Perfectly-secure MPC with linear communication complexity. In *Theory of Cryptography Conference (TCC)*, pages 213–230, 2008.

[7] M. Blanton. Achieving full security in privacy-preserving data mining. In *IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT)*, pages 925–934, 2011.

[8] P. Bunn and R. Ostrovsky. Secure two-party k-means clustering. In *ACM Conference on Computer and Communications Security (CCS)*, pages 486–497, 2007.

[9] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[10] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks (SCN)*, pages 182–199, 2010.

[11] O. Catrina and C. Dragulin. Multiparty computation of fixed-point multiplication and reciprocal. In *International Workshop on Database and Expert Systems Application (DEXA)*, pages 107–111, 2009.

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

[13] R. Cramer, I. Damgård, and U. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *Advances in Cryptology – EUROCRYPT*, pages 316–334, 2000.

[14] I. Damgård, M. Fitzi, E. Kiltz, J. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference (TCC)*, volume 3876 of *LNCS*, pages 285–304, 2006.

[15] I. Damgård, M. Geisler, and M. Krøigaard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography (IJACT)*, 1(1):22–31, 2008.

[16] I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Advances in Cryptology – EUROCRYPT*, pages 445–465, 2010.

[17] I. Damgård, Y. Ishai, M. Krøigaard, J. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *Advances in Cryptology – CRYPTO*, pages 241–261, 2008.

[18] P.-A. Fouque, J. Stern, and J.-G. Wackers. Cryptocomputing with rationals. In *Financial Cryptography (FC)*, pages 136–146, 2002.

[19] M. Franz, B. Deiseroth, K. Hamacher, S. Jha, S. Katzenbeisser, and H. Schröder. Secure computations on non-integer values. In *IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6, 2010.

[20] M. Franz and S. Katzenbeisser. Processing encrypted floating point signals. In *ACM Workshop on Multimedia and Security (MMSEC'11)*, pages 103–108, 2011.

[21] J. Garay, B. Shoenmakers, and J. Villegas. Practical and secure solutions for integer comparison. In *Conference on Theory and Practice of Public Key Cryptography (PKC)*, pages 330–342, 2007.

[22] R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 101–111, 1998.

[23] R.E. Goldschmidt. Applications of division by convergence. In *Master's thesis, M.I.T*, 1964.

[24] M. Hirt and U. Maurer. Robustness for free in unconditional multi-party computation. In *Advances in Cryptology – CRYPTO*, pages 101–118, 2001.

[25] T. Hoens, M. Blanton, and N. Chawla. A private and reliable recommendation system using a social network. In *IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT)*, pages 816–825, 2010.

[26] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *International Conference on Cryptology and Network Security (CANS)*, pages 1–20, 2009.

[27] Manuel Liedel. Secure distributed computation of the square root and applications. In *ISPEC*, pages 277–288, 2012.

[28] Y. Lindell and B. Pinkas. Privacy preserving data mining. *Journal of Cryptology*, 15(3):177–206, 2002.

[29] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit decomposition protocol. In *Conference on Theory and Practice of Public Key Cryptography (PKC)*, pages 343–360, 2007.

[30] K. Peng and F. Bao. An efficient range proof scheme. In *IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT)*, pages 826–833, 2010.

[31] T. Reistad. Multiparty comparison – An improved multiparty protocol for comparison of secret-shared values. In *International Conference on Security and Cryptography (SECRYPT)*, pages 325–330, 2009.

[32] T. Reistad and T. Toft. Linear, constant-rounds bit-decomposition. In *International Conference on Information, Security and Cryptology (ICISC)*, pages 245–257, 2009.

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

[34] T. Toft. Constant-rounds, almost-linear bit-decomposition of secret shared values. In *Topics in Cryptology – CT-RSA*, pages 357–371, 2009.

[35] A. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

A Additional Protocols

We describe an additional protocol which we call simple division SDiv. It is called as a sub-protocol from floating point division FLDiv and also serves the basis of our integer division implementation. SDiv follows the idea of FPDiv in [12] with two main differences. First, because our floating point representation maintains the values normalized, there is no need to call complex initial approximation AppRcr used in FPDiv, the main overhead of which comes from normalization. Instead, we compute the initial approximation w locally (line 2). The second difference is that we represent the current error term x as a pair of two ℓ-bit values x_1 and x_2. In an implementation of Goldschmidt's division algorithm x has a double precision compared to all other values, and we choose to split it instead of requiring the field size to grow in order to accommodate larger intermediate results. We therefore have $x = x_1 2^\ell + x_2$.

The algorithm is iterative (with θ iterations), and in each iteration we need to compute $y = y(\alpha + x)$ and $x = x^2$, where y stores the result and $\alpha = 2^{2\ell}$ is a constant. Because as a result of this computation the size of the intermediate values grows, we need to truncate the resulting values to ℓ bits for y and 2ℓ bits for x. Our idea is to perform truncations as we multiply, which prevents intermediate values from exceeding $2\ell + 1$ bits. In particular, we can write $y(2^{2\ell} + x) = (y(2^\ell + x_1) + (yx_2)/2^\ell)2^\ell$. In other words, we first multiply y by x_2, remove ℓ least significant bits of the 2ℓ-bit result, and then add the result to $2\ell + 1$-bit $y(2^\ell + x_1)$ (line 9 of SDiv). What remains is to truncate the computed values by $\ell + 1$ bits to produce new ℓ-bit y. The same idea is used for computing x^2 (lines 11–12) without increasing the size of the intermediate values beyond $2\ell + 1$. Lastly, to prepare for the next round, we compute the (x_1, x_2) representation of newly computed x (lines 13–14). After the last iteration, y is updated in the same way as before and returned as the result.

$[y] \leftarrow \mathsf{SDiv}([a],[b],\ell)$

```
 1. $(\theta, \alpha, \beta) \leftarrow (\lceil \log \frac{\ell}{3.5} \rceil, 2^{2\ell}, 2^{-\ell})$;
 2. $[w] \leftarrow 2.9142 \cdot 2^{\ell} - 2[b]$;
 3. $[x] \leftarrow \alpha - [b][w]$;
 4. $[y] \leftarrow [a][w]$;
 5. $[y] \leftarrow \mathsf{TruncPr}([y], 2\ell, \ell)$;
 6. $[x_2] \leftarrow \mathsf{Mod2m}([x], 2\ell, \ell)$;
 7. $[x_1] \leftarrow ([x] - [x_2])\beta$;
 8. for $i = 1$ to $\theta - 1$
 9.     $[y] \leftarrow [y]([x_1] + 2^{\ell}) + \mathsf{TruncPr}([y][x_2], 2\ell, \ell)$;
10.     $[y] \leftarrow \mathsf{TruncPr}([y], 2\ell + 1, \ell + 1)$;
11.     $[x] \leftarrow [x_1][x_2] + \mathsf{TruncPr}([x_2][x_2], 2\ell, \ell + 1)$;
12.     $[x] \leftarrow [x_1][x_1] + \mathsf{TruncPr}([x], 2\ell + 1, \ell - 1)$;
13.     $[x_2] \leftarrow \mathsf{Mod2m}([x], 2\ell, \ell)$;
14.     $[x_1] \leftarrow ([x] - [x_2])\beta$;
15. $[y] \leftarrow [y]([x_1] + 2^{\ell}) + \mathsf{TruncPr}([y][x_2], 2\ell, \ell)$;
16. $[y] \leftarrow \mathsf{TruncPr}([y], 2\ell + 1, \ell + 1)$;
17. return $[y]$;
```

Note that in the protocol above truncation is performed using function $\mathsf{TruncPr}([x], \ell, m)$ from [12] instead of previously described $\mathsf{Trunc}([x], \ell, m)$. The difference is that $\mathsf{TruncPr}$ has a significantly faster performance (2 rounds and $m + 1$ invocations, from which all but 1 round and 1 invocation can be precomputed), but the output is randomized. $\mathsf{TruncPr}$ rounds the result to the nearest integer with probability $1 - \delta$, where δ is the distance between $x/2^m$ and the nearest integer. $\mathsf{Mod2m}([x], \ell, m)$ computes $y = x \bmod 2^m$ and has exactly the same complexity as $\mathsf{Trunc}([x], \ell, m)$.