Alternative Implementations of Secure Real Numbers

Vassil Dimitrov Liisi Kerik Toomas Krips Jaak Randmets Jan Willemson

August 11, 2016

Abstract

This paper extends the choice available for secure real number implementations with two new contributions. We will consider the numbers represented in form $a - \varphi b$ where φ is the golden ratio, and in form $(-1)^s \cdot 2^e$ where e is a fixed-point number. We develop basic arithmetic operations together with some frequently used elementary functions. All the operations are implemented and benchmarked on Sharemind secure multi-party computation framework. It turns out that the new proposals provide viable alternatives to standard floating- and fixed-point implementations from the performance/error viewpoint in various settings. However, the optimal choice still depends on the exact requirements of the numerical algorithm to be implemented.

1 Introduction

It is estimated that currently human knowledge doubles approximately every year [21]. It means that data analysis facilities should keep up with this pace. However, it is not conceivable that all the organisations depending on big data analysis would upgrade their server farms every year.

Consequently, the only way to manage this growth is to rely on computation as a service, and this is the core reason behind the success of cloud computing business idea.

On the other hand, outsourcing computations has other restrictions with data privacy being on top of the list. Privacy-preserving data analysis is the holy grail of cloud computing, but unfortunately it is easier said than done.

There are several paradigms proposed enabling to offload some of the computations to another party in a way that some privacy guarantees could be given.

Historically, the first framework was the garbled circuits (GC) approach originally proposed by Yao in 1982 [3, 18, 23]. At its core, GC provides secure function evaluation capability for two mutually distrusting parties, one acting as a garbler and another as evaluator.

Soon, secure multi-party computation protocols were developed in a series of seminal papers [4, 9, 12]. These protocols can be implemented on top of several basic technologies, with secret sharing being one of the most often used ones [5, 22].

The most recent breakthrough in secure computation outsourcing was achieved by Gentry who proposed the first solution to achieve fully homomorphic encryption in 2009 [11].

All of these approaches have their strengths and weaknesses, but their common characteristic is that they introduce a remarkable performance penalty to achieve privacy-preserving features.

Archer *et al.* have compared the main secure computation implementations of AES-128 block cipher [2] and conclude that the fastest alternative is secure multi-party computing based on linear secret sharing.

However, an efficient secure computation framework is only the first step, providing basic elementary operations like bit manipulation or integer addition and multiplication. Statistical data analysis methods require higher level protocols like division and roots. To implement those, data types richer than bits and (modular) integers are required.

This paper presents a contribution on secure data types, more precisely, secure real number implementations.

The paper is organised as follows. First, we will review the state of the art in secure real number implementations in Section 2 and list the required preliminaries in Section 3. Then, Sections 4 and 5 describe two new proposed approaches based on golden section and logarithmic representations. Performance benchmarks and error analysis are presented in Section 6 and some conclusions are drawn in Section 7.

2 State of the art

The first attempt to extend modular integer domains provided by basic secure computation frameworks was made by Catrina *et al.* They implemented fixed-point arithmetic and applied it to linear programming [6–8].

In 2011, Franz and Katzenbeisser [10] proposed a solution to implementation of floating-point arithmetic for secure signal processing. Their approach relies on two-party computations working over garbled circuits, and they have not provided any actual implementation or benchmarking results.

In 2013, Aliasgari et al. designed and evaluated floating-point computation techniques and protocols for square root, logarithm and exponentiation in a standard linear secret sharing framework [1].

This approach was extended in 2015 by Kamm and Willemson who implemented and optimised a number of numeric primitives, building a complete solution for privacy-preserving satellite collision analysis [13].

In order to provide further improvements, Krips and Willemson [15, 16], and Kerik *et al.* [14] have proposed and implemented several tricks that increase the efficiency of specific elementary functions used under various restrictions.

The standardised IEEE 754 format is not well suited for secure computations on oblivious values. For example, it has an exceptional not-a-number value and it assumes explicit bit access that is inefficient to implement on top of modular integer arithmetic. Thus, all of the aforementioned implementations use a custom format, ignoring some of the details of IEEE 754 standard. The first full standard-compliant implementation was achieved by Pullonen and Siim who used a hybrid of garbled circuits and secret sharing [20]. However, the overhead required to support the whole standard is too high for larger data volumes.

3 Preliminaries

Our protocols will make use of both public and private (protected) values. To express that x is private, we will denote it by $[\![x]\!]$. Concrete instantiation of the value protection mechanism may vary between implementations. In principle, any secure computation framework (say, garbled circuits, fully homomorphic encryption or secret-sharing based multi-party computation) may be used.

However, we will make some assumptions about the underlying framework. We will assume that the framework provides access to modular integer arithmetic. In addition to integers, we use several kinds of fixed-point numbers.

Two's complement and biased fixed-point numbers will be represented and hence also notated as integers. Fixed-point numbers that use a sign bit instead of two's complement or biased representation will be denoted as tuples (s, a). A tuple (s, a) that is a signed fixed-point number with radix-point m signifies the value $(-1)^s \cdot a \cdot 2^{-m}$.

We will also assume access to the following operations.

- Addition of two private values $[\![x]\!]$ and $[\![y]\!]$ denoted as $[\![x]\!] + [\![y]\!]$. If the underlying protection framework is linear, this is equal to $[\![x + y]\!]$.
- Multiplication of a private value $[\![x]\!]$ by a public scalar c denoted as $c \cdot [\![x]\!]$. If the underlying protection framework is linear, this is equal to $[\![c \cdot x]\!]$.
- Multiplication of two private values $[\![x]\!]$ and $[\![y]\!]$ denoted as $[\![x]\!] \cdot [\![y]\!]$. If the underlying framework is not fully homomorphic, evaluating this primitive generally requires communication between the computing parties.

- Standard comparison operators $>$, $\geq$, etc. The inputs of these operators will be protected integer values and outputs will be protected bits containing the values of the corresponding predicate evaluations.
- Standard Boolean operations (conjunction, disjunction, xor) on one-bit protected values with the output being a protected bit again.
- `ObliviousChoice`($[\![b]\!], [\![x]\!], [\![y]\!]$): if the bit $b = 1$ then this function outputs $[\![x]\!]$, otherwise it outputs $[\![y]\!]$. Note that while usually x and y are integers, they might also refer to more complicated types that consist of several integer values.
- `Swap`($[\![c]\!], [\![x]\!], [\![y]\!]$) outputting $([\![x]\!], [\![y]\!])$ if $c = 0$ and $([\![y]\!], [\![x]\!])$ if $c = 1$. Here also x and y can refer to a type consisting of several integer values.
- `ConjBit`($\{[\![x_i]\!]\}_{i=0}^{n}, [\![y]\!]$) takes an array of bits x and a single bit y and finds the conjunction of every bit of x with y.
- `PublicBitShiftRightProtocol`($[\![x]\!], k$). This function Takes a protected value $[\![x]\!]$ and a public integer k, and outputs $[\![x \gg k]\!]$ where $x \gg k$ is equal to x shifted right by k bits. $x \gg k$ is also equal to $x/2^k$ rounded down, so sometimes in the protocol we will use syntactic sugar like $x/2^k$ to denote $x \gg k$.
- `BitExtract`($[\![x]\!]$) takes a protected value $[\![x]\!]$ and outputs a vector of protected bits corresponding to the bit representation of x. Note that when we use linear integer protection mechanism (like secret sharing) this operation is rather expensive.
- `MSNZB`($\{[\![b_i]\!]\}_{i=0}^{n-1}$) (Most Significant Non-Zero Bit) takes a vector of protected bits and outputs a similar vector, where only the highest 1-bit (i.e. the 1-bit with the largest index value) has remained and all the other bits are set to 0. If all the input bits are 0, they will also remain so in the output.
- Polynomial evaluation protocol `cPoly`($p, [\![x]\!]$) evaluates a public polynomial $p = \{p_i\}_{i=0}^{l}$ on $[\![x]\!]$. We consider that the inputs are fixed-point numbers and that both $[\![x]\!]$ and the output have 0 bits before radix point. Therefore, both the range and the domain are $[0, 1)$. Implementation details can be found in [14].
- Vectorised fixed-point polynomial evaluation protocol `cPolyArr`($p, [\![x]\!]$) takes an array of polynomials $p = \{\{p_{i,j}\}_{j=0}^{l}\}_{i=0}^{k}$ and an argument $[\![x]\!]$ and evaluates all the polynomials on $[\![x]\!]$. This protocol is similar to `cPoly` but more efficient than evaluating each polynomial independently.
- `Pick`($\{[\![x_i]\!]\}_{i=0}^{l}, [\![n]\!]$) takes a shared array $\{[\![x_i]\!]\}_{i=0}^{l}$ and a shared index $[\![n]\!]$ and returns $[\![x_n]\!]$.
- `Truncate`($[\![x]\!], n$) takes an integer $[\![x]\!]$ and casts it down to n bits by discarding the highest bits. We presume that the length of $[\![x]\!]$ is no less than n bits. If the underlying protection framework is linear then truncating a shared integer is achieved by truncating all the shares.
- `ConvertUp`($[\![x]\!], k, l$) takes in a shared k-bit two's complement integer $[\![x]\!]$ and returns a shared l-bit two's complement integer that has the same value. We presume that $k < l$.
- `FixSubtract`($([\![s_0]\!], [\![a_0]\!]), ([\![s_1]\!], [\![a_1]\!])$) takes two signed fixed-point numbers $([\![s_0]\!], [\![a_0]\!])$ and $([\![s_1]\!], [\![a_1]\!])$ and returns their difference as a signed fixed-point number.

For benchmarking, we will implement our algorithms on Sharemind[1] multi-party computation engine that relies on linear secret sharing and provides all the primitive operations listed above.

It was recently shown by Pettai and Laud that, when properly composed, Sharemind protocols provide privacy against active adversaries [19]. They also produced a software toolkit allowing for the respective analysis to run on the Sharemind protocols implemented in the domain specific language developed by Randmets and Laud [17]. All the protocols described in this paper have been formally verified using this toolkit.

[1]`https://sharemind.cyber.ee/`

4 Golden section numbers

We shall now describe a real number type that can depict signed real numbers, has free addition, and is reasonably efficient for other operations.

We use a tuple of secret signed two's complement integers $(\llbracket a \rrbracket, \llbracket b \rrbracket)$ to denote the positive real number $a - \varphi b$. We call these numbers *golden section numbers* or *golden numbers*. We may refer to these numbers as either $a - \varphi b$ or (a, b). For a given real number x, we denote its (approximate) golden representation as ${}^{g}x$. For a golden section number $a - \varphi b$, we refer to a as its integer representand and to b as its φ-representand. Note that we will be using a and b throughout this section to refer to the integer representand and the φ-representand of the number being considered, respectively.

We will now see how addition and multiplication work on golden section numbers. Addition is quite straightforward:

$$a - \varphi b + c - \varphi d = (a + c) - \varphi(b + d).$$

For multiplication, we note that $\varphi^2 = \varphi + 1$ and thus obtain

$$(a - \varphi b) \cdot (c - \varphi d) = (ac + bd) - \varphi(bc + ad - bd)\ . \tag{1}$$

Golden numbers are not monotone with respect to representands. Hence, finding a good approximation for a given number is a non-trivial problem on its own.

Definition 1. *Given a real number x, we say that the tuple of integers (a, b) is a (k, ε)-approximation of x if $|a|, |b| \leq 2^k$ and $|a - \varphi b - x| \leq \varepsilon$. If k is implied by the context or not important in the context, we shall refer to (a, b) as just an ε-representation of x. If ε is implied or not important, we shall refer to (a, b) as a $(k, \cdot)$-representation of x.*

If neither are important (or are implied) we refer to (a, b) as just as a representation of x.

It is preferable to use such a (k, ε)-approximation of a number where both k and ε are relatively small. While it is clear that a small ε implies a small error and is thus better, the reason why a small k is good is a bit more difficult.

Namely, we observe that when we multiply two golden section numbers x and y with $(k, \cdot)$-approximation, then their product has a $(2k + 1, \cdot)$-approximation. We will later see how we can replace a golden section number x with a (k, ϵ)-representation of it.

We shall assume throughout the section that the error ε is quite a small number, several orders of magnitude smaller than 1. Thus, when we discuss how either the number or the representands need to be bounded by some quite large numbers, we shall ignore ε in those analyses as rounding down used in finding those large numbers will cover any overflow ε might cause.

Golden numbers are also relatively dense in real numbers.

Lemma 1. *For a real number x which satisfies $|x| < \varphi^{s+1}$, and a positive integer k, there exists a $(\varphi^{s+1} + \varphi^{k+1}, \varphi^{-k})$-approximation of x.*

Proof. We note that we can write every positive real number as a (possibly infinite) sum of powers of φ. We can write $x = \sum_{i=-\infty}^{s} a_i \varphi^i$ where $a_i \in \{0, 1\}$ and where there is no i so that $a_i = 1$ and $a_{i+1} = 1$ would both hold.

We also note that given such a requirement, $\sum_{i=-\infty}^{s'} a_i \varphi^i < \varphi^{s'+1}$ holds for any s'.

Thus, if we choose to represent x as $x = \sum_{i=s'}^{s} a_i \varphi^i$, the error we make is no greater than $\varphi^{s'}$, that is, $|x - \sum_{i=s'}^{s} a_i \varphi^i| \leq \varphi^{s'}$.

The following three facts about Fibonacci numbers F_k $(k \in \mathbb{Z})$ are common knowledge and are easily proven:

- $\varphi^k = F_k \varphi + F_{k-1}$ for every k,
- $F_k \approx \varphi^k$ for every positive k, and

- $\sum_{i=0}^{k} F_k = F_{k+1} - 1$ for every positive k.

From these facts it follows that

$$\sum_{i\ =s'}^{s} a_i\varphi^i = \sum_{i=s'}^{s} a_i(F_i\varphi + F_{i-1}) \leq \sum_{i=s'}^{s}|F_i|\varphi + |F_{i-1}|=$$
$$\sum_{i=0}^{s'}|F_i|\varphi + |F_{i-1}|+\sum_{i=1}^{s}|F_i|\varphi + |F_{i-1}|=$$
$$(F_{s'+1} - 1)\varphi + (F_{s'}) + (F_{s+1} - 2)\varphi + F_s - 1$$
$$= (F_{s'+1} + F_{s+1} - 3)\varphi + F_s + F_{s'} - 1.$$

We see that taking $k = -s'$ gives us the result. □

However, there is a problem with this number system. Namely, we may use large integers to depict small real numbers. This, however, means that when multiplying several small real numbers, the representands may grow exponentially and may overflow very fast. We would like to keep the absolute value of the representands to be smaller than some reasonable constant.

The solution for this comes from the fact that there may be several different (k, ε)-approximations of a number. Thus we want a method for replacing a (k_1, ε_1)-approximation with a (k_2, ε_2)-approximation where ε_2 may be slightly greater than ε_1, but where $k_2 < k_1$. We shall use a method that we shall call *normalization*, which is, in essence, subtracting a suitable representation of 0 from the golden section number.

Definition 2. *We say that a golden section number is ℓ-normalized if the absolute value of its integer representand is not greater than ℓ.*

Note that this definition depends only on the integer representand and not the φ-representand of a golden number, and that a too large φ-representand could also cause similar overflow problems as a too large integer representand does.

However, we shall see that if the absolute value of integer representand is smaller than ℓ then, provided that we put an extra constraint on the size of the numbers that we can represent, the absolute value of the φ-representand shall also be smaller than ℓ. More precisely, we shall require that $|a - \varphi b|\leq \ell(\varphi - 1)$.

Lemma 2. *Let a golden section number $a - \varphi b$ satisfy also $|a - \varphi b|\leq \ell(\varphi - 1)$ and be ℓ-normalized. Then $|b|\leq \ell$.*

Proof. Using the reverse triangle inequality, we obtain $||a| - |b\varphi|| \leq |a - b\varphi| \leq \ell(\varphi-1)$. From this we obtain $|b\varphi| - \ell(\varphi - 1) \leq |a| \leq |b\varphi| + \ell(\varphi - 1)$. Consider the first half — $|b\varphi|-\ell(\varphi - 1) \leq |a|$, from which we obtain $|b\varphi|-\ell\varphi + \ell \leq \ell$, i.e. $|b\varphi| \leq \ell\varphi$. This is equivalent to $|b| \leq \ell$. □

We thus will assume from this point on that if we are talking about a golden section number, then $|a - \varphi b|\leq \ell(\varphi - 1)$.

We want to settle for a general value for ℓ that we will use across the article. The main idea for choosing ℓ shall be the idea that if we multiply together two ℓ-normalized numbers, no overflow should happen.

We shall generally take $\ell = \left\lfloor\sqrt{\frac{2^{n-1}-1}{2}}\right\rfloor$, where n refers to the bit length of a and b.

Lemma 3. *If two golden section numbers $a - \varphi b$ and $c - \varphi d$ are $\left\lfloor\sqrt{\frac{2^{n-1}-1}{2}}\right\rfloor$-normalized, then both the integer representand and the φ-representand of their product are smaller than 2^{n-1}.*

Proof. Let us denote $a - \varphi b \cdot c - \varphi d$ with $x - \varphi y$, i.e. $x = ac + bd$ and $y = ad + bc - bd$. We give the proof for $x \geq 0$ as the proof for $x < 0$ is analogous. We assume that $|a|, |b|, |c|, |d| \leq \ell$. Thus $x = ac+bd \leq 2\ell^2 < 2^{n-1}$. We assume that the absolute value of the product $x - \varphi y$ is no greater than $\ell(\varphi - 1)$. In a similar way as we did in the proof of Lemma 2, we obtain that $|y\varphi|-\ell(\varphi - 1) \leq x$. Thus $|y\varphi|\leq 2\ell^2 + \ell(\varphi - 1)$ which gives us $|y|< 2\ell^2 < 2^{n-1}$. □

From now on, we shall speak of normalized numbers which shall mean $\left\lfloor \sqrt{\frac{2^{n-1}-1}{2}} \right\rfloor$-normalized numbers. Likewise, ℓ shall refer to $\left\lfloor \sqrt{\frac{2^{n-1}-1}{2}} \right\rfloor$.

4.1 Normalization

In the previous subsection we described a problem where representands grow exponentially on multiplication and threaten to overflow really fast.

Thus, we want to significantly reduce the absolute values of the representands of the number while keeping its value relatively the same.

There are various possibilities for this, but due to the nature of the task, they are equivalent to deducing suitable $\bar{\varepsilon}$-representations of zero from the number. Namely, suppose that we normalized $a - \varphi b$ and ended up with $a' - \varphi b'$. Let the error made in this process be no greater than $\bar{\varepsilon}$, i.e. $|a - \varphi b - (a' - \varphi b')| \leq \bar{\varepsilon}$. This means that $|(a-a')-\varphi(b-b')| \leq \bar{\varepsilon}$, i.e. $|(a - a') - \varphi(b - b')|$ is an $\bar{\varepsilon}$-representation of 0. The smaller the $\bar{\varepsilon}$, the smaller the error arising from the normalization process and thus it is desirable to obtain $\bar{\varepsilon}$-representations of 0 where $\bar{\varepsilon}$ is very small. The process should also be not very resource-consuming.

We first note that, thanks to Lemma 2, it suffices to normalize only the integer representand of the number. If the normalization error is small, then the result will still be an ε-representation of a golden number, and thus the absolute value of the φ-representand of the result will also be smaller than ℓ.

Note also that in order to normalize an integer representand down to 2^k, we need either the $n - k$ most significant bits to be zero (if this repesentand is positive) or the $n - k$ most significant bits to be one (if it is negative) in the end-result. (In principle, k can be chosen freely, but we generally take $k = \frac{n}{2} - 1$.) We note that using the protocol $\mathsf{BitExtract}([\![\cdot]\!])$, we can have access to the individual bits of a.

If the bit representation of a is $a_0 a_1 \ldots a_{n-1}$ then $a = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$ (recall we are using two's complement representation). In the positive case we could make the i-th bit of a zero by deducting 2^i from a. We will later confirm that in the negative case in a similar fashion we could make the ith bit zero by adding 2^i to a. In the end we would use a_{n-1} to obliviously choose between the negative and positive cases.

Thus we shall describe a method where for every bit a_i of a we have a representation of 0 that is $(2^i, \left[\frac{2^i}{\varphi}\right])$, i.e. $2^i - \varphi\left[\frac{2^i}{\varphi}\right]$. We would perform the $\mathsf{BitExtract}([\![a]\!])$ protocol and then use the higher bits of a to perform oblivious choice about whether to deduce or add $(2^i, \left[\frac{2^i}{\varphi}\right])$ from a or not. In the positive case we would end up with $a - \sum_{i=k}^{n-2} a_i 2^i = \sum_{i=0}^{k-1} a_i 2^i \leq 2^k - 1$ and in the negative case with

$$\begin{aligned}
a + \sum_{i=k}^{n-1}(1-a_i)2^i &= -2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i + \sum_{i=k}^{n-2}(1-a_i)2^i \\
&= -1 - \sum_{i=0}^{n-2} 2^i + \sum_{i=0}^{n-2} a_i 2^i + \sum_{i=k}^{n-2}(1-a_i)2^i \\
&= -1 + \sum_{i=0}^{n-2}(a_i - 1)2^i + \sum_{i=k}^{n-2}(1-a_i)2^i \\
&= -1 + \sum_{i=0}^{k-1}(a_i - 1)2^i \\
&\geq -1 + \sum_{i=0}^{k-1} -2^i = -2^k.
\end{aligned}$$

We would then perform oblivious choice between them using a_{n-1}. We would end up with a number where the absolute value of a is no greater than 2^k which is small enough for a suitable k.

However, this approach has a problem – namely, $(2^i, \left[\frac{2^i}{\varphi}\right])$ tends to be a very poor representation of 0. The set $\{|k - \varphi[\frac{k}{\varphi}]| | k \in \mathbb{Z}\}$ is quite uniformly distributed in $[0, \frac{\varphi}{2})$ and thus for most integers k, $(k - \varphi\left[\frac{k}{\varphi}\right])$ is a poor representation of 0 as the error is too large.

Thus we want to modify our algorithm somewhat — we want to find numbers x_i that are close to 2^i but where $|x_i - \varphi\left[\frac{x_i}{\varphi}\right]|$ is very small. On the other hand, we also wish $|2^i - x_i|$ to be small, since if $|2^i - x_i|$ is too large, then the number that we obtain after the normalization process is still not normalized.

We shall now describe the algorithm for normalization. We use a number of pairs of public constants (x_i, y_i) such that $x_i - \varphi y_i \approx 0$ and that x_i is close to 2^i. After describing the algorithm we shall see what properties they must satisfy. The protocol is formalized in Algorithm 1.

We are given a golden section number $[\![a]\!] - \varphi[\![b]\!]$. We perform bit decomposition on $[\![a]\!]$ and obtain its bits $[\![a_0]\!], \dots, [\![a_{n-1}]\!]$. Out of these, we are interested in bits with large indices as the less significant bits will not be important in normalizing. Let us consider the positive and negative cases separately for easier reading. In the algorithm, we will compute both of them and then use a_{n-1} to obliviously choose between them. First let us consider the case where a is positive. If $a_i = 1$ and $n - 1 > i \geq k$, we will deduce x_i from a and y_i from b. This is done by multiplying x_i with $[\![a_i]\!]$ and deducing $[\![x_i a_i]\!]$ from $[\![a]\!]$, and likewise, multiplying y_i with $[\![a_i]\!]$ and deducing $[\![y_i a_i]\!]$ from $[\![b]\!]$. Note that these protocols are local and thus practically free.

Likewise, in the negative case, if $a_i = 0$ and $n - 1 > i \geq k$, we will add x_i to a and y_i to b.

Algorithm 1: `GoldenNorm`

```
   Data: $[\![a]\!], [\![b]\!], \{x_i\}_{i=k}^{n}, \{y_i\}_{i=k}^{n}$
   Result: Given a golden section number and a normalization set, returns the number normalized
           according to the set.
 1 $\{[\![a_i]\!]\}_{i=0}^{n-1} \leftarrow \mathsf{BitExtract}([\![a]\!])$;
 2 $\{[\![z_i]\!]\}_{i=k}^{n-2} \leftarrow \{[\![a_i]\!]\}_{i=k}^{n-2} \cdot \{x_i\}_{i=k}^{n-2}$;
 3 $\{[\![w_i]\!]\}_{i=k}^{n-2} \leftarrow \{[\![a_i]\!]\}_{i=k}^{n-2} \cdot \{y_i\}_{i=k}^{n-2}$;
 4 $\{[\![z'_i]\!]\}_{i=k}^{n-2} \leftarrow \{[\![1 - a_i)]\!]\}_{i=k}^{n-2} \cdot \{x_i\}_{i=k}^{n-2}$;
 5 $\{[\![w'_i]\!]\}_{i=k}^{n-2} \leftarrow \{[\![(1 - a_i)]\!]\}_{i=k}^{n-2} \cdot \{y_i\}_{i=k}^{n-2}$;
 6 for $i \leftarrow k$ to $n - 2$ do
 7     $[\![a']\!] \leftarrow [\![a]\!] - [\![z_i]\!]$;
 8     $[\![b']\!] \leftarrow [\![b]\!] - [\![w_i]\!]$;
 9     $[\![a'']\!] \leftarrow [\![a]\!] + [\![z'_i]\!]$;
10     $[\![b'']\!] \leftarrow [\![b]\!] + [\![w'_i]\!]$;
11 end
12 $[\![a]\!] \leftarrow \mathtt{ObliviousChoice}([\![a_{n-1}]\!], [\![a']\!], [\![a'']\!])$;
13 $[\![b]\!] \leftarrow \mathtt{ObliviousChoice}([\![a_{n-1}]\!], [\![b']\!], [\![b'']\!])$;
14 return $[\![a]\!], [\![b]\!]$
```

Now we shall see what properties the pairs (x_i, y_i) must satisfy so that the final result would have an absolute value no greater than ℓ and that its difference from the original golden number would be no greater than ϵ.

We want the end result, which is $a - \sum_{i=k}^{n-2} a_i x_i$ in the positive case and $a + \sum_{i=k}^{n-2}(1 - a_i)x_i$ in the negative case, to be in the interval $(-\ell, \ell)$. We note that in the positive case the following equality holds.

$$a - \sum_{i=k}^{n-2} a_i x_i = \sum_{i=0}^{k-1} a_i 2^i + \sum_{i=k}^{n-2} a_i (2^i - x_i).$$

Likewise, in the negative case this holds.

$$a + \sum_{i=k}^{n-2}(1 - a_i)x_i = -1 + \sum_{i=0}^{k-1}(a_i - 1)2^i + \sum_{i=k}^{n-2}(a_i - 1)(2^i - x_i).$$

In attempting to estimate these quantities with inequalities, it is important whether 2^i is smaller or greater than x_i. Thus, by distinguishing these cases, we arrive at the following inequalities:

$$\begin{aligned}\sum_{\substack{i:2^i<x_i\\ k\le i\le n-2}} (2^i - x_i) &\le \sum_{i=0}^{k-1} a_i 2^i + \sum_{i=k}^{n-2} a_i(2^i - x_i)\\ &\le 2^k - 1 + \sum_{\substack{i':2^{i'}>x_i\\ k\le i\le n-2}} (2^{i'} - x_{i'}) \ ,\end{aligned}$$

$$\begin{aligned}\sum_{\substack{i:2^i<x_i\\ k\le i\le n-2}} (x_i - 2^i) &\ge -1 + \sum_{i=0}^{k-1} (a_i - 1)2^i + \sum_{i=k}^{n-2} (1 - a_i)(x_i - 2^i)\\ &\ge -2^k + \sum_{\substack{i:2^i>x_i\\ k\le i\le n-2}} (x_i - 2^i) \ .\end{aligned}$$

Thus, in order to achieve that $a - \sum_{i=k}^{n-2} a_i x_i$ or $a + \sum_{i=k}^{n-2} (1 - a_i)x_i$ belongs the interval $(-\ell, \ell)$, it suffices for both cases that

$$-\ell \le \sum_{\substack{i:2^i<x_i\\ k\le i\le n-2}} (2^i - x_i)$$

and

$$2^k + \sum_{\substack{i':2^{i'}>x_i\\ k\le i\le n-2}} (2^{i'} - x_{i'}) \le \ell \ .$$

Thus we arrive to the following definition.

Definition 3. *A $(k, \ell, \varepsilon, n)$-normalization set is a set of integers $\{x_k, \ldots, x_{n-1}, y_k, \ldots, y_{n-1}\}$ with the following properties:*

1. $\sum_{i=k}^{n-2} |x_i - \varphi \cdot y_i| \le \varepsilon$
2. $\sum_{\substack{i:2^i<x_i\\ k\le i\le n-2}} (2^i - x_i) \ge -\ell$
3. $2^k + \sum_{\substack{i':2^{i'}>x_i\\ k\le i\le n-2}} (2^{i'} - x_{i'}) \le \ell$

There is some freedom in choosing k, with lower values giving us more freedom in choosing the normalization set, but reducing the number of values that a normalized number can have.

4.2 Finding Normalization Sets

The key part of finding normalization sets is finding integer pairs (x_i, y_i) where on the one hand $\frac{x_i}{y_i}$ is very close to φ and on the other hand x_i must be very close to 2^i. In essence, we have to find suitable constants x_i, since y_i is defined by x_i as $[\frac{x_i}{\varphi}]$. Note that this can be considered a minimization problem – we have to guarantee that properties 2 and 3 in Definition 3 hold, but how loosely or strictly they hold is not really important for us.

On the other hand, we do want to minimize $\sum_{i=k}^{n-1} |x_i - \varphi \cdot y_i|$ as much as possible in order to minimize the error caused by the normalization protocol. Thus we can see finding the normalization set as an optimisation problem to minimize $\sum_{i=k}^{n-1} |x_i - \varphi \cdot [\frac{x_i}{\varphi}]|$ with the constraints 2. and 3. in Definition 3 holding.

For easier notation, let us define $\mathrm{err}(x) := x - \varphi[\frac{x}{\varphi}]$.

We searched for these coefficients using the facts that $\mathrm{err}(F_k)$ tend to be small and that when when x and x' are such that if $\mathrm{err}(x)$ and $\mathrm{err}(x')$ are small, then also $\mathrm{err}(x + x')$ is small, more precisely, $|\mathrm{err}(x + x')| \leq |\mathrm{err}(x)| + |\mathrm{err}(x')|$.

Thus, we would take a small interval around a power of 2 and find all elements z_i for which $\mathrm{err}(z_i)$ was suitably small. Then, in a larger interval, the only possible candidates w for a minimal $|\mathrm{err}(w)|$ had to have the format $z_i + j \cdot F_k$. Thus we needed to check these elements to find a minimal one. If necessary, this process could be iterated.

4.3 Protocols for golden section numbers

We shall now present a few protocols on golden numbers. We have already described addition, multiplication and normalization protocols and thus we will not discuss them any further here.

We will denote with $\texttt{GoldenMult}(\llbracket x \rrbracket, \llbracket y \rrbracket)$ golden number multiplication as described in equation (1). Generally we assume that all functions get normalized inputs, unless specified otherwise. We will thus not normalize the inputs before performing multiplication, but will normalize the product. In some cases, such as when it is the final result that will be declassified, the product can be left unnormalized.

We will also use the function $\texttt{GoldenProd}(x_0, \ldots, x_{k-1})$ to refer to computing the product $\prod_{i=0}^{k-1} x_i$ using $\texttt{GoldenMult}$. Computing the product of $k - 1$ golden numbers takes $l \cdot \log k$ rounds where l is the number of rounds required for a single multiplication.

4.3.1 Multiplication by φ

We will describe now a protocol for multiplying an integer by the golden ratio. This protocol, presented in Algorithm 2, will be useful for performing golden-to-fix conversion described in Section 4.3.2.

Algorithm 2: `MultWithPhi`

```
   Data: $\llbracket x \rrbracket, n, m, (m > n), \{p_i\}_{i=0}^{\infty}$
   Result: $\llbracket x\varphi \rrbracket$.
1  $\{\llbracket x_i \rrbracket\}_{i=0}^{n-1} \leftarrow \mathsf{BitExtract}(\llbracket x \rrbracket)$;
2  $\llbracket s_0 \rrbracket \leftarrow \sum_{i=0}^{m} p_i \cdot (\sum_{j=0} \llbracket x_j \rrbracket \cdot 2^{m+j-i})$;
3  $\llbracket s_1 \rrbracket \leftarrow \sum_{i=m+1}^{m+n} p_i \cdot (\sum_{j=i-m} \llbracket x_j \rrbracket \cdot 2^{m+j-i})$;
4  $\llbracket s \rrbracket \leftarrow \llbracket s_0 \rrbracket + \llbracket s_1 \rrbracket$;
5  $\{\llbracket x'_i \rrbracket\}_{i=0}^{n-1} \leftarrow \mathsf{BitExtract}(\llbracket -x \rrbracket)$;
6  $\llbracket s'_0 \rrbracket \leftarrow \sum_{i=0}^{m} p_i \cdot (\sum_{j=0} \llbracket x'_j \rrbracket \cdot 2^{m+j-i})$;
7  $\llbracket s'_1 \rrbracket \leftarrow \sum_{i=m+1}^{m+n} p_i \cdot (\sum_{j=i-m} \llbracket x'_j \rrbracket \cdot 2^{m+j-i})$;
8  $\llbracket s' \rrbracket \leftarrow \llbracket s'_0 \rrbracket + \llbracket s'_1 \rrbracket$;
9  $\llbracket r \rrbracket \leftarrow \texttt{ObliviousChoice}(\llbracket x_{n-1} \rrbracket, \llbracket s' \rrbracket, \llbracket s \rrbracket)$;
10 return $(\llbracket x_{n-1} \rrbracket, \llbracket r \rrbracket)$
```

The protocol takes in a secret signed integer $\llbracket x \rrbracket$ and returns a signed fixed-point number that represents $\llbracket x\varphi \rrbracket$. This protocol needs one bit-extraction protocol and one oblivious choice. We start with a secret integer $\llbracket x \rrbracket$. We also have the bits of ϕ, $\{p_i\}_{i=0}^{\infty}$. We extract the bits $\llbracket x_i \rrbracket$ from the input $\llbracket x \rrbracket$. We then compute $\sum_{i=0}^{m} p_i \cdot (\sum_{j=0} \llbracket x_j \rrbracket \cdot 2^{m+j-i}) + \sum_{i=m+1}^{m+n} p_i \cdot (\sum_{j=i-m} \llbracket x_j \rrbracket \cdot 2^{m+j-i})$ that represents $x\varphi$ if x is positive. We then do the same for $-x$ and obliviously choose between the two cases based on the last bit of x. The last bit of x is also the sign of the resulting fixed-point number, as multiplication with φ does not change the sign.

4.3.2 Conversion to a fixed-point number

Algorithm 3 presents the protocol for converting a golden section number to a fixed-point number.

Algorithm 3: `GoldToFix`

```
Data: $[\![a]\!] - \varphi[\![b]\!], n, m, (n > m)$
Result: A fixed-point number that represents the same value as the golden number input.
1 $[\![bigA]\!] \leftarrow$ ConvertUp($[\![a]\!], n, n + m$);
2 // we will also obtain $a_{n-1}$ as a side product from the ConvertUp function.
3 $[\![fixA]\!] \leftarrow ([\![a_{n-1}]\!], [\![bigA]\!] \cdot 2^m)$;
4 $[\![fixB]\!] \leftarrow$ MultWithPhi($[\![b]\!], n, m$);
5 $[\![fix]\!] \leftarrow$ FixSubtract($[\![fixA]\!], [\![fixB]\!]$);
6 return $[\![fix]\!]$
```

While conversion functions are important on their own, here we will also use them as subprotocols in more complicated algorithms.

Since we have access to `MultWithPhi` function, converting a golden number to a fixed-point number is trivial. We need to convert both the integer representand and the φ-representand to a respective fixed-point number and deduce the second from the first.

4.3.3 Return a constant based on power of two

We will see that in both the inverse protocol and the square root protocol, we get a secret golden number $[\![^g x]\!]$ and, based on the interval $[2^i, 2^{i+1})$ its absolute value is in, return a golden number $[\![^g z_i]\!]$.

The protocol for performing this operation is presented in Algorithm 4.

Algorithm 4: `TwoPowerConst`

```
Data: $[\![^g x]\!], \{^g z_i\} = \{(x_i, y_i)\}, n, m < n$
Result: Will return the sign of the input and $(x_j, y_j)$ if $^g x \in [2^j, 2^{j+1})$.
1 $[\![sign]\!], [\![f]\!] \leftarrow$ GoldToFix($[\![^g x]\!]$);
2 $\{[\![b_i]\!]\}_{i=0}^{n+m-1} \leftarrow$ MSNZB($[\![f]\!]$);
3 $([\![s]\!], [\![t]\!]) \leftarrow \sum_{i=-m}^{n-1}([\![b_i]\!] \cdot x_i, [\![b_i]\!] \cdot y_i)$;
4 return $[\![sign]\!], ([\![s]\!], [\![t]\!])$
```

The computation is performed the following way. We convert the input to a fixed-point number. We then perform `MSNZB` on the integer representative of the fixed-point number and compute the scalar product with the set of public coefficients $\{^g z_i\}$. Note that finding the scalar product is a local operation.

4.3.4 Inverse

We shall now describe the protocol for computing the inverse of a secret golden number $[\![^g x]\!]$. A protocol for computing the inverse of numbers in $[0.5, 1]$ is presented in Algorithm 5. It uses the approximation $\frac{1}{x} = \prod((1-x)^{2^i} + 1)$ that works well in the neighbourhood of 1 (being equivalent to the respective Taylor series).

The protocol for computing the inverse of a golden number is presented in Algorithm 6.

We use Algorithm 5 as a subprotocol. Given $^g x$ as an input, we need to find $^g x'$ and $^g y$ so that $x' \in [0.5, 1]$ and that $x \cdot y = x'$. We can then use the `HalfToOneInv` function to compute $\frac{1}{x'} = \frac{1}{x} \cdot \frac{1}{y}$ which we shall then multiply with y to obtain $\frac{1}{x}$. y is an approximation of a suitable power of 2.

We compute y using the function `TwoPowerConst`($[\![^g x]\!], \{^g z_i\}$). Here the $^g z_i$ are approximations of different powers of 2 – when $x \in [2^j, 2^{j+1})$, then `TwoPowerConst` should return approximately 2^{-j-1}.

We compute $[\![^g x']\!]$ by multiplying $[\![^g x]\!]$ and $[\![^g y]\!]$. We then use the `HalfToOneInv` protocol on $[\![^g x']\!]$, obtaining $[\![^g \frac{1}{x'}]\!]$. To get this back to the correct range, we multiply it by $[\![^g y]\!]$.

Algorithm 5: `HalfToOneInv`

```
Data: $[\![^g x]\!](x \in [0.5, 1]), n, m, (n > m), k$
Result: $[\![^g \frac{1}{x}]\!]$
1  $[\![^g y]\!] \leftarrow 1 - [\![^g x]\!]$;
2  $[\![^g y_0]\!] \leftarrow [\![^g y]\!]$;
3  for $i \leftarrow 0$ to $k-1$ do
4      $[\![^g y_{i+1}]\!] \leftarrow$ GoldenMult($[\![^g y_i]\!], [\![^g y_i]\!]$);
5  end
6  $[\![^g z]\!] \leftarrow$ GoldenProd($[\![^g y_0]\!] + 1, [\![^g y_1]\!] + 1, \ldots, [\![^g y_k]\!] + 1$);
7  return $[\![^g z]\!]$
```

Algorithm 6: `GoldInv`

```
Data: $[\![^g x]\!], n, m, (n > m), \{(x_i, y_i)\}$
Result: $[\![^g \frac{1}{x}]\!]$
1  $([\![sign]\!], [\![^g y]\!]) \leftarrow$ TwoPowerConst($[\![^g x]\!], \{^g z_i\}$);
2  $[\![^g x']\!] \leftarrow$ GoldenMult($[\![^g x]\!], [\![^g y]\!]$);
3  $[\![^g z]\!] \leftarrow$ HalfToOneInv($[\![^g x']\!]$);
4  $[\![^g w]\!] \leftarrow$ GoldenMult($[\![^g y]\!], [\![^g z]\!]$);
5  $[\![^g u]\!] \leftarrow$ ObliviousChoice($[\![sign]\!], -[\![^g w]\!], [\![^g w]\!]$);
6  return $[\![^g u]\!]$
```

Finally, since our current result is approximately $[\![^g \left|\frac{1}{x'}\right|]\!]$, we have to make an oblivious choice between the result and its additive inverse so that it would have the correct sign.

4.3.5 Square Root Protocol

Algorithm 7 presents the protocol for finding the square root of a golden section number.

Algorithm 7: `GoldSqrt`

```
Data: $[\![^g x]\!], m, n, (n > m), k, \{^g w_i\}$
Result: $[\![^g \sqrt{x}]\!]$
1  $[\![^g y_0]\!] \leftarrow$ TwoPowerConst($[\![^g x]\!], \{^g w_i\}$);
2  for $i \leftarrow 0$ to $k-1$ do
3      $[\![^g z_0]\!] \leftarrow$ GoldenMult($[\![^g y_i]\!], [\![^g x]\!]$);
4      $[\![^g z_1]\!] \leftarrow$ GoldenMult($[\![^g y_i]\!], [\![^g z_0]\!]$);
5      $[\![^g z_1]\!] \leftarrow 3 - [\![^g z_1]\!]$;
6      $[\![^g z_2]\!] \leftarrow$ GoldenMult($[\![^g y_i]\!], ^g 0.5$);
7      $[\![^g y_{i+1}]\!] \leftarrow$ GoldenMult($[\![^g z_1]\!], [\![^g z_2]\!]$);
8  end
9  $[\![^g w]\!] \leftarrow$ GoldenMult($[\![^g x]\!], [\![^g y_k]\!]$);
10 return $[\![^g w]\!]$
```

The protocol is following. We first compute the inverse square root of the input x and then multiply it with x. There exists an iterative algorithm for $\frac{1}{\sqrt{x}}$ where the formula for the nth approximation is $y_{n+1} = 0.5y_n(3 - xy_n^2)$. The reason why we use inverse square root to compute square root is that general iterative methods for square root need division, which is too costly in out setting.

To obtain the starting approximations, we shall use the function `TwoPowerConst` where the constants are $2^{\frac{i}{2}}$ – if $x \in [2^j, 2^{j+1})$, the function will return $2^{\frac{j}{2}}$.

5 Logarithmic numbers

In this section we will present logarithmic numbers. We will explain the format, and then describe algorithms for computing the inverse, product, square root, logarithm, exponential function and sum.

5.1 Logarithmic number format

We represent a logarithmic number $\mathbf{x}$ as a triple (z_x, s_x, e_x). Zero-bit z_x is 0 if $\mathbf{x}$ is zero and 1 if $\mathbf{x}$ is non-zero. Sign bit s_x is 0 if $\mathbf{x}$ is positive and 1 if $\mathbf{x}$ is negative. Exponent e_x is an $(m+n)$-bit integer which represents a fixed-point number with m bits before and n bits after the radix point. The exponent is biased and so can be both positive and negative. The value of the number is computed as follows: $(z, s, e) \to z \cdot (-1)^s \cdot 2^{(e-\texttt{Bias})/2^n}$, where `Bias` is $2^{m+n-2} - 1$. The larger m, the larger the range of numbers we can represent. The larger n, the more precision we have.

While the length of the exponent is $m + n$ bits, only the lowest $m + n - 1$ bits are used. The highest bit is always zero to achieve faster comparisons between exponents.

5.2 Inverse

Algorithm 8 presents the protocol for finding the inverse of a logarithmic number.

Algorithm 8: `LogNumInv`

```
  Data: $[\![\mathbf{x}]\!] = ([\![z_x]\!], [\![s_x]\!], [\![e_x]\!])$
  Result: $[\![1/\mathbf{x}]\!]$
1 return $([\![1]\!], [\![s_x]\!], 2 \cdot [\![\texttt{Bias}]\!] - [\![e_x]\!])$
```

Inverse is computed by leaving the sign unchanged and negating the exponent, based on the formula $((-1)^{s_x} \cdot 2^{e_x})^{-1} = (-1)^{s_x} \cdot 2^{-e_x}$. We assume that the input is not zero. The zero-bit of the result is set to 1 to indicate that the result is non-zero. We also have to account for the bias when computing the exponent. When changing the sign of a biased integer, we have to not only change the sign but also add the double of the bias.

5.3 Multiplication

Algorithm 9 presents the protocol for multiplying logarithmic numbers.

Algorithm 9: `LogNumMult`

```
  Data: $[\![\mathbf{x}]\!] = ([\![z_x]\!], [\![s_x]\!], [\![e_x]\!]), [\![\mathbf{y}]\!] = ([\![z_y]\!], [\![s_y]\!], [\![e_y]\!])$
  Result: $[\![\mathbf{x} \cdot \mathbf{y}]\!]$
1 $[\![e]\!] \leftarrow [\![e_x]\!] + [\![e_y]\!]$;
2 $[\![z]\!] \leftarrow [\![z_x]\!] \wedge [\![z_y]\!] \wedge ([\![e]\!] \geq [\![\texttt{Bias}]\!])$;
3 return $([\![z]\!], [\![s_x]\!] \oplus [\![s_y]\!], [\![e]\!] - [\![\texttt{Bias}]\!])$
```

Multiplication of logarithmic numbers is based on the formula $2^{e_x} 2^{e_y} = 2^{e_x + e_y}$. Because our exponents are biased, we have to subtract the bias when adding them. To get the sign of the result, we compute the XOR of the signs of the operands. The zero-bit of the end result is computed as follows: the result is non-zero iff both operands are non-zero and their product does not underflow. Therefore, any underflows are rounded down to zero.

Algorithm 10: `LogNumSqrt`

```
Data: $[\![\mathbf{x}]\!] = ([\![z_x]\!], [\![s_x]\!], [\![e_x]\!])$
Result: $[\![\sqrt{\mathbf{x}}]\!]$
1 return $([\![z_x]\!], [\![0]\!], ([\![e_x]\!] + [\![\texttt{Bias}]\!])/2)$
```

5.4 Square root

Algorithm 10 presents the protocol for finding the square root of a logarithmic number.

We assume that the input is non-negative. If the input is zero we return zero, and if the input is non-zero then we divide the exponent by two because $\sqrt{2^{e_x}} = 2^{e_x/2}$. When dividing a biased integer by two we have to double the bias before division in order to get the end result with the correct bias.

5.5 Logarithm

Algorithm 11 presents the protocol for finding the binary logarithm of a logarithmic number.

Algorithm 11: `LogNumLg`

```
Data: $m, n, [\![\mathbf{x}]\!] = ([\![z_x]\!], [\![s_x]\!], [\![e_x]\!]), p = \{p_i\}_{i=0}^{l}$
Result: $[\![\lg \mathbf{x}]\!]$
1 $[\![s]\!] \leftarrow [\![e_x]\!] < [\![\texttt{Bias}]\!]$;
2 $[\![e']\!] \leftarrow [\![e_x]\!] - [\![\texttt{Bias}]\!]$;
3 $[\![e'']\!] \leftarrow [\![\texttt{Bias}]\!] - [\![e_x]\!]$;
4 $[\![e]\!] \leftarrow \texttt{ObliviousChoice}([\![s]\!], [\![e'']\!], [\![e']\!])$;
5 $\{[\![j_i]\!]\}_{i=0}^{m+n-3} \leftarrow \texttt{MSNZB}(\texttt{BitExtract}([\![e]\!]))$;
6 $[\![v]\!] \leftarrow [\![e]\!] - \sum_{i=0}^{m+n-3} [\![j_i]\!] \cdot 2^i$;
7 $[\![w]\!] \leftarrow \sum_{i=0}^{m+n-3} ([\![j_i]\!] \cdot 2^{m+n-i}) \cdot [\![v]\!]$;
8 $[\![z]\!] \leftarrow [\![e_x]\!] \neq [\![\texttt{Bias}]\!]$;
9 $[\![t]\!] \leftarrow \texttt{cPoly}(p, [\![w]\!])/2^{m-1}$;
10 $[\![u]\!] \leftarrow 2^{n+1} \cdot \sum_{i=0}^{m+n-3} ([\![j_i]\!] \cdot (n+1-i))$;
11 return $([\![z]\!], [\![s]\!], [\![t]\!] - [\![u]\!] + [\![\texttt{Bias}]\!])$
```

To compute the binary logarithm of a logarithmic number, we assume that the input is positive. We note that if $2^{e_y} = \lg 2^{e_x}$ then $e_y = \lg e_x$. Therefore, the exponent of the output is the binary logarithm of the exponent of the input, which means that the problem is reduced to computing the binary logarithm of a fixed-point number. However, logarithmic numbers with negative and zero exponents (logarithmic numbers that lie in $(0, 1]$) need special handling, because we do not want to deal with computing the logarithms of negative numbers. If the exponent of the input is negative, we find the result using the formula $\lg 2^{-e_x} = -2^{\lg e_x}$. Thus, to compute the binary logarithm of a logarithmic number we compute the logarithm of the absolute value of the exponent, and set the sign bit of the end result to 0 if the exponent is positive, and 1 if the exponent is negative. If the exponent is equal to 0 then we set the zero-bit of the result to 0, otherwise we set it to 1.

We compute the binary logarithm of a fixed-point number with the help of a polynomial p that approximates $f'(x) = \log_4(x+1) + 1/2$ in $[0, 1)$ (obtained using Chebychev interpolation). Our polynomial evaluation protocol only allows inputs and outputs in range $[0, 1)$, therefore, instead of approximating $f(x) = \lg x$ directly, we first shift the number to the left so that we shift out all leading zeroes and the most significant non-zero bit (via `BitExtract` and `MSNZB` protocols and multiplications). Then we consider the resulting number as a fixed-point number with 0 bits before radix point and approximate the function $f'(x) = \log_4(x+1) + 1/2$, the values of which are in $[0.5, 1)$. In order to derive the logarithm of the original number from this intermediate result, we divide it by 2^{m-1} and subtract a constant which depends on the most significant non-zero bit of the original number.

In order to compute natural logarithm via binary logarithm, we multiply the result by $\ln 2$, because $\log_a x = \log_a 2 \cdot \lg x$.

5.6 Exponent

Algorithm 12 presents the protocol for finding the base 2 exponent of a logarithmic number.

```
Algorithm 12: LogNumExp
  Data: $m, n, [\![\mathbf{x}]\!] = ([\![z_x]\!], [\![s_x]\!], [\![e_x]\!]), p = \{p_i\}_{i=0}^{l}$
  Result: $[\![2^{\mathbf{x}}]\!]$
1 $[\![g]\!] \leftarrow \texttt{Truncate}([\![e_x]\!], n)$;
2 $[\![w]\!] \leftarrow [\![2^{m-2} + m - 3]\!] - [\![e_x]\!]/2^n$;
3 $[\![y]\!] \leftarrow \texttt{cPoly}(p, 2^{m-2} \cdot [\![g]\!])$;
4 $[\![t]\!] \leftarrow \texttt{Pick}(\{[\![y]\!]/2^i\}_{i=0}^{m+n-3}, [\![w]\!])$;
5 $[\![z]\!] \leftarrow \overline{[\![z_x]\!]} \vee ([\![e_x]\!] < [\![2^n \cdot (m-2) + \texttt{Bias}]\!])$;
6 $[\![u]\!] \leftarrow [\![z_x]\!] \cdot \texttt{ObliviousChoice}([\![s_x]\!], -[\![t]\!], [\![t]\!]) + [\![\texttt{Bias}]\!]$;
7 return $([\![z]\!], [\![0]\!], [\![u]\!])$
```

To compute the base 2 exponential function on logarithmic numbers, we note that if $2^{e_y} = 2^{2^{e_x}}$ then $e_y = 2^{e_x}$. Therefore, the exponent of the output is 2 to the power of the exponent of the input, and the problem is reduced to computing base 2 exponential function on fixed-point numbers.

To find the base 2 exponent of a fixed-point number, we use the formula $2^{e_x} = 2^{\lfloor e_x \rfloor + 1} \cdot 2^{\{e_x\} - 1}$. Separating the input into whole and fractional part allows us to approximate the function $f(x) = 2^{x-1}$ on the fractional part, with inputs in range $[0, 1)$ and outputs in range $[0.5, 1)$, which is suitable for our interpolation protocol. In the algorithm, approximation polynomial for $f(x) = 2^{x-1}$ is denoted as p.

After exponentiating the fractional part, we shift the result right by a number of bits which depends on the whole part. This is done by computing all public shifts and obliviously picking the right one (line 4).

Note that in order to achieve good precision when approximating $2^{\{e_x\}-1}$, we do not just truncate the exponent to get the fractional part. We truncate and then perform a left shift so that polynomial evaluation protocol has more bits of precision to work with. Therefore, after polynomial evaluation we do not have to perform a left shift, which shifts in zeroes from the right and therefore means lost precision, but a right shift, which shifts out bits to the right and therefore means ridding ourselves of excess precision achieved with polynomial approximation.

In order to compute $\exp x$, we multiply the argument by $\lg e$, because $a^x = 2^{\lg a \cdot x}$.

5.7 Addition

Algorithm 13 presents the protocol for finding the sum of logarithmic numbers.

First, we sort the operands by their absolute value. If the absolute value of $\mathbf{x}$ is smaller than the absolute value of $\mathbf{y}$ then we swap them (lines 1 and 2). Now we know that $|\mathbf{x}| \geq |\mathbf{y}|$. In order to reduce addition to a single-operand function, we factorise $2^{e_x} \pm 2^{e_y}$ as $2^{e_x}(1 \pm 2^{e_y - e_x}) = 2^{e_x + \lg(1 \pm 2^{e_y - e_x})}$. Knowing that the first operand is larger is beneficial for two reasons: it gives us the sign of the end result (which is equal to the sign of the larger operand), and it ensures that $\lg(1 - 2^{e_y - e_x})$ is a real-valued function. Now that the operands are sorted, we approximate two different functions, one for addition and one for subtraction.

To compute $\lg(1 \pm 2^{e_y - e_x})$ we find the difference of the exponents (line 3) and denote it with e. We also find all its bits (line 4). We denote with t (line 5) a bit which is 0 if we perform addition and 1 if we subtract.

Both $f_a(x) = \lg(1 + 2^x)$ and $f_d(x) = \lg(1 - 2^x)$ are functions for which interpolating with a single polynomial though Chebyshev nodes yields poor precision, especially for the values of the argument near

Algorithm 13: `LogNumAdd`

```
Data: $[\![\mathbf{x}]\!] = ([\![z_x]\!], [\![s_x]\!], [\![e_x]\!]), [\![\mathbf{y}]\!] = ([\![z_y]\!], [\![s_y]\!], [\![e_y]\!]), a = \{a_i\}_{i=0}^{m+n-3}, d = \{d_i\}_{i=0}^{m+n-3}, c = 2^n(2^m + \lg(1 - 2^{-2^{-n}}))$
Result: $[\![\mathbf{x} + \mathbf{y}]\!]$
1  $[\![l]\!] \leftarrow \overline{[\![z_x]\!]} \vee [\![z_y]\!] \wedge ([\![e_x]\!] < [\![e_y]\!])$;
2  $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!]) \leftarrow \texttt{Swap}([\![l]\!], [\![\mathbf{x}]\!], [\![\mathbf{y}]\!])$;
3  $[\![e]\!] \leftarrow [\![e_x]\!] - [\![e_y]\!]$;
4  $\{[\![b_i]\!]\}_{i=0}^{m+n-1} \leftarrow \texttt{BitExtract}(e)$;
5  $[\![t]\!] \leftarrow [\![s_x]\!] \oplus [\![s_y]\!]$;
6  $\{[\![p_i]\!]\}_{i=0}^{m+n-2} \leftarrow \texttt{MSNZB}(\{[\![b_i]\!]\}_{i=0}^{m+n-2})$;
7  $\{[\![r_i]\!]\}_{i=0}^{m+n-2} \leftarrow \texttt{ConjBit}(\{[\![p_i]\!]\}_{i=0}^{m+n-2}, [\![t]\!])$;
8  $\{[\![q_i]\!]\}_{i=0}^{m+n-2} \leftarrow \{[\![r_i]\!]\}_{i=0}^{m+n-2} \oplus \{[\![p_i]\!]\}_{i=0}^{m+n-2}$;
9  $[\![k]\!] \leftarrow \sum_{i=1}^{m+n-2}([\![p_i]\!] \cdot 2^{m+n-1-i} \cdot ([\![e]\!] - [\![2^i]\!]))$;
10 $\{[\![v_i]\!]\}_{i=0}^{m+n-3} \leftarrow \texttt{cPolyArr}(a, [\![k]\!])$;
11 $\{[\![w_i]\!]\}_{i=0}^{m+n-3} \leftarrow \texttt{cPolyArr}(d, [\![k]\!])$;
12 $[\![g'_a]\!] \leftarrow (\overline{\bigvee_{i=0}^{m+n-2}[\![b_i]\!]} \oplus [\![q_0]\!]) \cdot 2^n$;
13 $[\![g_a]\!] \leftarrow \sum_{i=0}^{m+n-3}([\![q_{i+1}]\!] \cdot [\![v_i]\!])$;
14 $[\![g'_d]\!] \leftarrow [\![r_0]\!] \cdot c$;
15 $[\![g_d]\!] \leftarrow \sum_{i=0}^{m+n-3}([\![r_{i+1}]\!] \cdot [\![w_i]\!])$;
16 $[\![u]\!] \leftarrow [\![z_y]\!] \cdot ([\![g'_a]\!] + [\![g_a]\!] + [\![g'_d]\!] + [\![g_d]\!])$;
17 $[\![z]\!] \leftarrow [\![z_x]\!] \oplus [\![z_x]\!] \wedge [\![z_y]\!] \wedge [\![t]\!] \wedge ([\![e_x]\!] < -[\![u]\!])$;
18 return $([\![z]\!], [\![s_x]\!], [\![e_x]\!] + [\![u]\!])$
```

zero. Therefore, our approach for approximating these functions is to find the most significant non-zero bit of the argument (line 6) and shift the argument to the left so that we shift out all leading zeroes and the most significant non-zero bit (line 9). On the resulting number we compute a separate polynomial for each function and each possible position of the most significant non-zero bit (lines 10 and 11). (In the algorithm, we denote the array of polynomials for addition as a and the array of polynomials for subtraction as d.)

There are also two special cases which are not covered by the polynomials: if the argument is zero, and if the most significant non-zero bit of the argument is in the lowest possible position. If the argument is zero then it means that $\mathbf{x}$ and $\mathbf{y}$ have an equal absolute value, in which case for addition we return as the value of $f_a(x) = \lg(1 + 2^x)$ a constant representing 1 (line 12), and for subtraction we return 0 as the final result. If the most significant non-zero bit of the argument is in the lowest possible position then for addition we return as the value of $f_a(x) = \lg(1 + 2^x)$ a constant representing 1, and for subtraction we return as the value of $f_d(x) = \lg(1 - 2^x)$ a constant c representing $2^m + \lg(1 - 2^{-2^{-n}})$ (line 14).

In lines 13 and 15 we pick the right interpolation result depending on the most significant nonzero bit of the argument. From this collection of constants and polynomial approximation results, we pick the correct one based on whether we are performing addition or subtraction, and depending on the most significant non-zero bit of $[\![e_x]\!] - [\![e_y]\!]$. In line 16 we pick the value u which is added to the exponent of the larger logarithmic number to achieve the final result. Note that if the smaller operand is zero then u is also set to zero.

In case of subtraction, we check for underflow, and if the result of subtraction is smaller than is possible to accurately represent with a non-zero number we round the result down to zero (line 17).

The approach to addition presented in Algorithm 13 results in precision which is reasonable but still far from ideal. One way to perform precise logarithmic addition is based on formula $\mathbf{x}+\mathbf{y} = \mathbf{x} \cdot \lg(2 \cdot 2^{\mathbf{y}/\mathbf{x}})$ where we find the sum of two numbers with the help of division, exponentiation, multiplication, and logarithm. In order to achieve good precision with this method, the operands have to be cast up before computing the sum. As this method involves a composition of exponentiation and logarithm, both performed on numbers twice the size of the original inputs, it is extremely inefficient, but it allows for near-ideal precision.

6 Analysis of error and performance

In this section, we compare golden and logarithmic numbers against existing implementations of floating-point and fixed-point representations. Comparisons are made in both performance and accuracy for different operations and bit-widths. We look at addition, multiplication, reciprocal and square root. For floating-point and logarithmic numbers, we additionally measure the performance and accuracy of exponentiation and natural logarithm.

We have implemented logarithmic and golden section numbers on the SHAREMIND SMC platform. We chose SHAREMIND because of its maturity, tooling, and availability of fixed-point and floating-point numbers. As existing number systems were already implemented using SHAREMIND's domain-specific language [17], we decided to also use it for golden section and logarithmic representations. The protocol language provides us with directly comparable performance and allows to avoid many complexities that a direct C++ implementation would have.

To provide a clear overview of accuracy and speed trade-offs, we measured the performance of each number system on multiple bit-widths. Generally, higher bit-widths offer us better accuracy for the cost of performance.

We implemented three different versions of logarithmic numbers: $\mathcal{L}_h$, $\mathcal{L}_s$ and $\mathcal{L}_d$ (with h, s and d standing for *half*, *single* and *double* precision). For $\mathcal{L}_h$, we chose $m = 6$ and $n = 16$ (Section 5.1) so that it offers at least as much range and precision as IEEE 754 half-precision floating-point numbers and also aligns to a byte boundary ($2 + 6 + 16 = 24$ bits or 3 bytes). For $\mathcal{L}_s$, we chose $m = 9$ and $n = 29$ for a size of 40 bits and accuracy comparable to single-precision floating-point numbers. For $\mathcal{L}_d$, we chose $m = 12$ and $n = 58$ for a size of 72 bits and accuracy comparable to double-precision floating-point numbers.

We also implemented tree versions of golden numbers (Section 4): $\mathcal{G}_{32}$, $\mathcal{G}_{64}$ and $\mathcal{G}_{128}$ where for $\mathcal{G}_n$ we store two n-bit components to provide comparable accuracy to n-bit fixed-point numbers with radix point at $\lfloor n/2 \rfloor$.

We compare our results against existing secure real number implementations that SHAREMIND already provides. Two floating-point number representations are used: float_s, providing comparable accuracy to single-precision floating-point numbers, and float_d, providing comparable accuracy to double-precision floating-point numbers. See [13–15] for implementation details. Logarithmic numbers compare well with floating-point numbers as both are designed to provide good relative errors. Additionally, SHAREMIND provides 32-bit and 64-bit fixed-point numbers with radix point in the middle (denoted with fix_{32} and fix_{64} respectively). Golden numbers compare well with fixed-point numbers as both are designed to provide good absolute errors.

Accuracy was measured experimentally, by identifying the range of inputs in which the largest errors should be found, and then uniformly sampling this range to find maximum error.

Performance measurements were made on a cluster of three computers connected with 10Gbps Ethernet. Each cluster node was equipped with 128GB DDR4 memory and two 8-core Intel Xeon (E5-2640 v3) processors, and was running Debian 8.2 Jessie with memory overcommit and swap disabled.

We measured each operation on various input sizes, executing the operation in parallel on the inputs. Each measurement was repeated a minimum of ten times and the mean of the measurements was recorded. Measurements were performed in randomized order. Note that due to the networked nature of the protocols, parallel execution improves performance drastically up to the so called *saturation point*.

We recorded maximum achieved *operations per second* that states how many parallel operations can be evaluated on given input size per second. For example, if we can perform a single parallel operation on 100-element vectors per second this gives us 100 operations per second.

Our performance and accuracy measurements are displayed in Figure 1. For every variant of every real number representation, we plot its maximum achieved performance in operations per second (OP/s) on the y-axis and its error on the x-axis. Note that performance increases on the y-axis and accuracy improves on the x-axis. Logarithmic numbers are represented with squares, golden section with diamonds, floating-point numbers with circles and fixed-point numbers with triangles. The accuracy of operations increases with shade, so that white shapes denote least precision and best performance.

We have measured addition, multiplication, reciprocal and square root. For floating-point and logarithmic numbers we also benchmarked exponentiation and natural logarihm. In most cases, maximum relative error was measured, but for fixed-point numbers and some golden number protocols this is not reasonable. In these cases maximum absolute error was measured, and this is denoted by adding label "A" to the mark.

Some operations, such as fixed-point addition, achieve perfect accuracy within their number representation. These cases are marked with a "$\star$". Instead of maximum error, we plot the value of half of the step between two consecutive numbers in this representation

In Figure 1 we can see that golden section numbers compare relatively well with fixed-point numbers. They achieve somewhat worse performance in our aggregated benchmarks, but the true picture is actually more detailed.

What is not reflected on the graph is the fact that golden section multiplication requires significantly fewer communication rounds than fixed-point multiplication. For instance, fix_{32} multiplication requires 16 communication rounds, but comparable $\mathcal{G}_{64}$ multiplication requires only 11. This makes golden section numbers more suitable for high latency and high throughput situations, and also better for applications that perform many consecutive operations on small inputs.

The worse performance of golden section numbers after the saturation point can be wholly attributed to increased communication cost. Namely, every multiplication of $\mathcal{G}_{64}$ numbers requires 6852 bits of network communication, but a single fix_{32} multiplication requires only 2970.

We can also see that compared to floating-point numbers, logarithmic numbers perform significantly better in case of multiplication (Figure 1b), reciprocal (Figure 1d) and square root (Figure 1c), while offering similar accuracy. Unfortunately, logarithmic numbers do not compare favourably to floating-point numbers with both lower performance and worse accuracy of addition (Figure 1a). In case of natural logarithm and exponentiation, logarithmic numbers are close to floating-point numbers in both performance and accuracy. This means that logarithmic numbers are a poor choice for applications that are very addition heavy but an excellent choice for applications that require many multiplicative operations.

7 Conclusions and further work

Technically, protected computation domains are very different from the classical open ones. Many low-level bit manipulation techniques are too cumbersome to implement and hence standard numeric algorithms do not work very well.

This holds true even for basic arithmetic operations. A full IEEE 754 floating-point number specification is too complex to be efficient in an oblivious setting. Even a reimplementation of the $significand \cdot 2^{exponent}$ representation is too slow, even in case of simple addition, since oblivious radix point alignment is very inefficient. Hence, alternatives need to be studied.

This paper proposed two new candidate representations for oblivious real numbers – golden and logarithmic representations. The corresponding algorithms were implemented on the Sharemind SMC engine and benchmarked for various precision levels and input sizes.

The results show that we still do not have a clear winner.

Since logarithmic representation is multiplicative, adding two logarithmic numbers is slow. However, significant performance improvements can be achieved for several elementary functions like multiplication, inverse, and square root.

Golden number representation allows for very fast (actually, local) addition, and its multiplication speed is comparable with that of fixed-point numbers. However, this format only allows for relatively slow elementary function computations.

Thus the choice of real number representation depends on application domain and computations to be performed.

Another aspect to consider is precision. Our analysis shows that logarithmic representation achieves the best relative error for most of the operations (except addition). However, precision achieved by our other implementations seems more than sufficient for practical statistical applications.

In this paper we have developed only the most basic mathematical tools. In order to be applied to actual data analysis tasks (e.g. statistical tests, finding correlation coefficients, variances, etc.), higher-level operations need to be implemented. It is an interesting open question which real number representations perform optimally for various operations and input sizes. This study will be a subject for our future research.

8 Acknowledgements

This research has been supported by Estonian Research Council under the grant no. IUT27-1.

References

[1] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS*, 2013.

[2] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and performance of programmable secure computation. Cryptology ePrint Archive, Report 2015/1039, 2015. `http://eprint.iacr.org/`. Journal version accepted to IEEE Security & Privacy, to appear in 2017.

[3] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[4] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1988.

[5] George Robert Blakley. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, 1979.

[6] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks*, pages 182–199. Springer, 2010.

[7] Octavian Catrina and Sebastiaan De Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In *Computer Security–ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 134–150. Springer, 2010.

[8] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography and Data Security*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2010.

[9] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19. ACM, 1988.

[10] Martin Franz and Stefan Katzenbeisser. Processing encrypted floating point signals. In *Proceedings of the thirteenth ACM multimedia workshop on Multimedia and security*, pages 103–108. ACM, 2011.

[11] Craig Gentry. *A fully homomorphic encryption scheme.* PhD thesis, Stanford University, 2009.

[12] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[13] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, 14(6):531–548, 2015.

[14] Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. LNCS. Springer, 2016. Accepted to Workshop on Applied Homomorphic Cryptography 2016.

[15] Toomas Krips and Jan Willemson. Hybrid model of fixed and floating point numbers in secure multiparty computations. In *Information Security: 17th International Conference, ISC 2014*, volume 8783 of *LNCS*, pages 179–197. Springer, 2014.

[16] Toomas Krips and Jan Willemson. Point-counting method for embarrassingly parallel evaluation in secure computation. In *FPS 2015*, volume 9482 of *LNCS*, pages 66–82. Springer, 2016.

[17] Peeter Laud and Jaak Randmets. A Domain-Specific Language for Low-Level Secure Multiparty Computation Protocols. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015*, pages 1492–1503. ACM, 2015.

[18] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[19] Martin Pettai and Peeter Laud. Automatic Proofs of Privacy of Secure Multi-party Computation Protocols against Active Adversaries. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 75–89. IEEE, 2015.

[20] Pille Pullonen and Sander Siim. Combining Secret Sharing and Garbled Circuits for Efficient Private IEEE 754 Floating-Point Computations. In *FC 2015 Workshops*, volume 8976 of *LNCS*, pages 172–183. Springer, 2015.

[21] David Russell Schilling. Knowledge doubling every 12 months, soon to be every 12 hours. *Industry Tap*, 2013. `http://www.industrytap.com/knowledge-doubling-every-12-months-soon-to-be-every-12-hours/3950`.

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

[23] Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

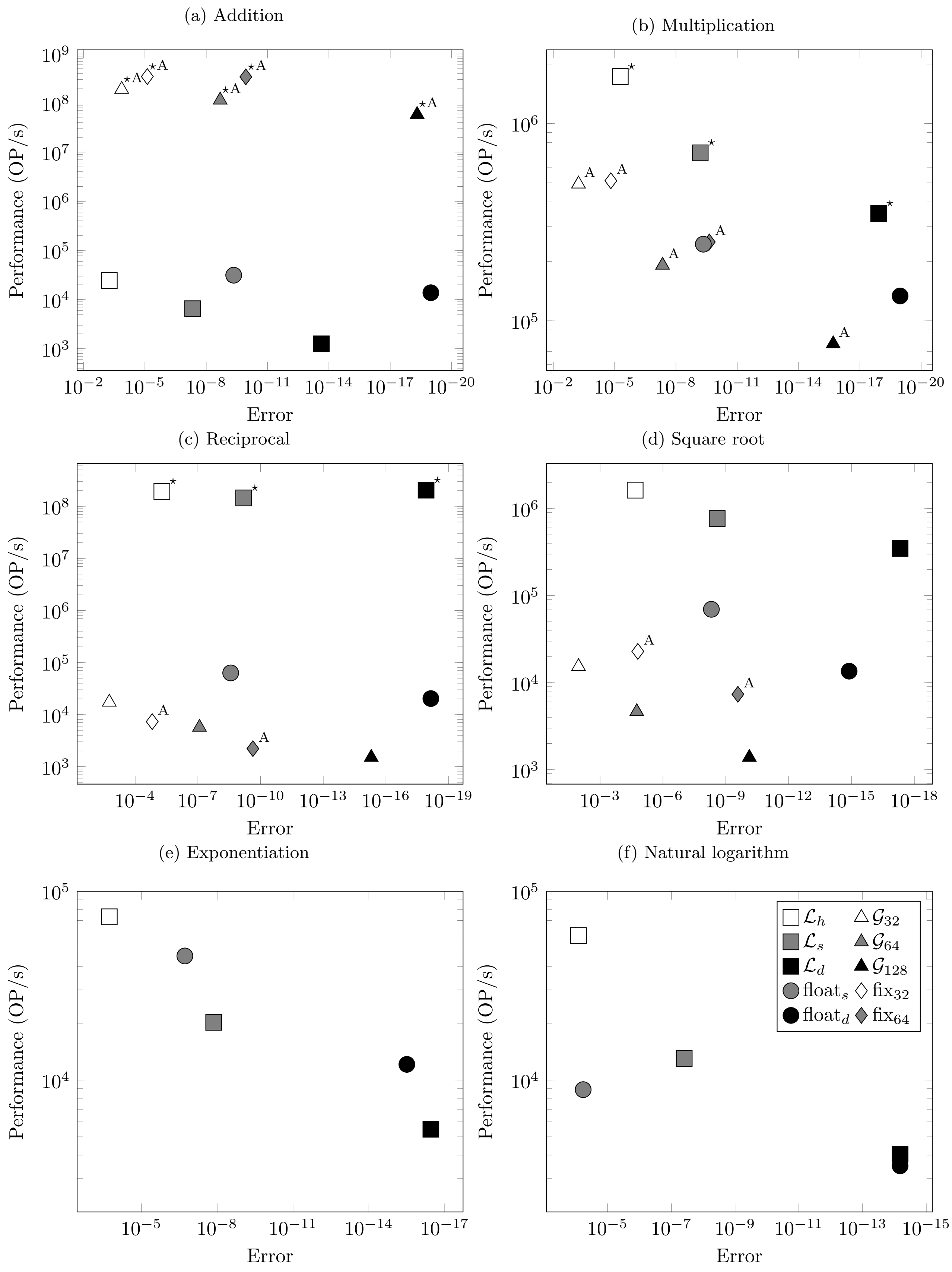

Figure 1: Performance and accuracy trade-offs between various real number representations. Performance in operations per second is denoted on the y-axis and absolute or relative error on the x-axis. In most cases maximum relative error was measured but in a few cases maximum absolute error was measured – we denote this using "A". Performance increases on the y-axis and accuracy improves on the x-axis. We annotate operations that achieve perfect accuracy within the representation with $\star$ and plot the value of half of the step between two consecutive numbers in this representation.