# Benchmarking Privacy Preserving Scientific Operations

Abdelrahaman Aly[1][0000−0003−2038−5668] and Nigel P. Smart[1,2][0000−0003−3567−3304]

[1] imec-COSIC, KU Leuven, Leuven, Belgium.
[2] University of Bristol, Bristol, UK.
abdelrahaman.aly@esat.kuleuven.be, nigel.smart@kuleuven.be

**Abstract.** In this work, we examine the efficiency of protocols for secure evaluation of basic mathematical functions (`sqrt`, `sin`, `arcsin`, amongst others), essential to various application domains. e.g., Artificial Intelligence. Furthermore, we have incorporated our code in state-of-the-art Multiparty Computation (MPC) software, so we can focus on the algorithms to be used as opposed to the underlying MPC system. We make use of practical approaches that, although, some of them, theoretically can be regarded as less efficient, can, nonetheless, be implemented in such software libraries without further adaptation. We focus on basic scientific operations, and introduce a series of data-oblivious protocols based on fixed point representation techniques. Our protocols do not reveal intermediate values and do not need special adaptations from the underlying MPC protocols. We include extensive computational experimentation under various settings and MPC protocols.

## 1 Introduction

Secure Multiparty Computation (MPC) allows a set of parties to compute an arbitrary function of their inputs without revealing anything about them, except for what can be deduced from the output of the function. Standard MPC protocols usually provide secure basic operations, such as additions, multiplications and logic gates, from which more complex functionalities can be built. In many data processing applications one requires access to standard scientific operations, and thus to an approximation of what in the `C` language is represented by the data types `float` and `double`. There are two techniques for performing this approximation in the literature: fixed point representations and floating point representations. Both of which have been implemented in various MPC systems; for example [2, 6, 10].

Efficient algorithms for fixed point representations were introduced in a series of works by Catrina et al (see [8] for a detailed summary). Typically, fixed point arithmetic uses a publicly available, predefined precision to which all data values are kept. This is more efficient than a floating point representation, as examined in [1] for example, where the operations are closer to what one would expect for the equivalent `C` data types. However, the increased cost of using floating point representations makes their use highly problematic in practice. In addition, it is sufficient, in many applications, to work with a fixed point precision, for example in various statistical or machine learning applications.

Despite many prior works on fixed point operations in MPC, there has been little work on benchmarking these operations. In this work we provide a number of benchmarks of simple scientific operations on fixed point numbers in a variety of cases. In particular, we focus on standard mathematical operations such as $\texttt{sqrt}(x), \texttt{sin}(x), \texttt{cos}(x), \texttt{arcsin}(x), \texttt{exp}(x), \texttt{log}_a(x)$, etc, in both the full threshold setting with two and three players, and in the honest majority three party setting. We build our protocols on top of the actively-secure-with-abort MPC protocols available in the `SCALE-MAMBA` system [2]. This system is chosen as it is publicly available and allows different access structures to be utilized for the same input program.

A number of previous works have looked at *algorithms* to implement these operations [1, 3, 4, 11, 21, 22]; however our paper concentrates on investigating practical performance and in addition looks at optimizations to such protocols in the case of MPC systems which allow a certain amount of pre-processing (for example in `SCALE-MAMBA` shared random bits are produced in the pre-processing, and so come for free in the 'online' phase).

One can consider the current state of the art of MPC development as mirroring the development of standard computer programming and architectures. Thus, one often needs to go an revisit earlier works to understand simple ways of implementing functions which might be taken for granted. In this work we make extensive use of the methods used in the 1960s and 1970s to build mechanisms to evaluate scientific operations on fixed and floating point data. In particular we use the work on function approximation by Taylor and Padé approximations provided in the seminal work of Hart [16].

In our work a secret fixed point value is represented by a secret integer $v$ in the range $[-2^{k-1}, \ldots, 2^{k-1}]$ ($k$ is fixed and public) and a public precision value $f$. The fixed point number represented by $v$ is $v/2^f$. In the kind of MPC systems we consider (i.e. those based on linear secret sharing) the value $v$ is embedded in a finite field $\mathbb{F}_q$. We shall denote by $[\![v]\!]$ the fact that $v$ is shared over $\mathbb{F}_q$. To represent a fixed point number $x$, whose sharing we denote by $\langle x \rangle$, we take an integer $v$ in the above range such that $x \approx v/2^f$. Thus we can write $\langle x \rangle = \{[\![v]\!], k, f\}$.

Given secure protocols for addition and multiplication of $\mathbb{F}_q$ elements, one can construct secure protocols for addition and multiplication of fixed point values, see [8] for the precise protocols. These protocols will be secure as long as the underlying protocols for addition and multiplication of $\mathbb{F}_q$ elements are also secure.

The addition and multiplication protocols of [8] often require one to open 'masked' values of fixed point values. To do this, there is a statistical security parameter $\kappa$, and we require that $\kappa + 2 \cdot k < \log_2 q$. The statistical security parameter measures the statistical distance of certain opened values in the protocol from a uniform distribution. In particular, when we want to open a masked $k$ or

$2 \cdot k$ bit integer value, we use a mask of $k + \kappa$ or $2 \cdot k + \kappa$ bits. This ensures that the distribution of the masked output is within statistical distance of $2^{-\kappa}$ from the uniform distribution. In our implementation we take $(\log_2 q, \kappa, k) = (128, 40, 41)$ and $f = 20$.

We emphasize that our work is centered around MPC based on secret sharing based-MPC, as opposed to systems based on Garbled Circuits or Homomorphic Encryption. More specifically, our works is aimed at any set of individuals that want to perform these operations, whilst using some LSSS based system. Clearly, on this regard, there are different tradeoffs and algorithms if the underlying MPC system chosen is one based on, say Garbled Circuits, however, this falls outside the scope of this work.

**Square Root Function:** The problem of computing securely the `sqrt` has been studied by several authors in both the floating point and fixed point settings e.g., [1, 18, 21, 22]. The collection of work that addressed this problem before, approached it by either expressing the output as a Taylor series, or via some other kind of numerical approximation (e.g. via Goldschmidt or Newton-Raphson approximation).

Liedel's [22] method at first glance seems to be the most efficient. This method assumes a fixed point representation in line to what is expressed in [8], produces (given private input $\langle x \rangle$) first an initial approximation of $\sqrt{\langle x \rangle}$. This initial approximation is then improved by performing iterations of Goldschmidt and Newton-Raphson. Liedel offers a way to calculate the initial approximation by solving a system of equations over normalized inputs. However, there is a hidden assumption in the method, which turns out to be very restrictive in practice. This makes the applicability of Liedel's method less useful, and has motivated modern multiparty frameworks, such as Sharemind to use Taylor series based protocols [18]. In our method we use the spirit of Liedel's method, i.e. we use Goldshmidt and Newton-Raphson to perform the final approximation, however we produce the initial approximation to the $\sqrt{(\langle x \rangle)}$ by computing its closest power of two.

**Trigonometric functions:** There are a number of works that explore alternatives to build trigonometric functions [3, 4]. All use numerical approximations or series evaluations, but restrict the values to specific ranges (i.e. angle reduction is not performed before the trigonometric function is computed). We on the other hand, offer an angle reduction protocol that is designed to take advantage of the fixed point representation of [8]. This would naively utilize a division with remainder operation which is usually a more expensive primitive [1, 21], than a multiplication. Instead, we make an intelligent use of the fixed point representation of our inputs and a series of more basic operations. We then utilize, as do the two prior works, the numerical methods of Hart [16] to produce the final approximation to $\texttt{sin}(x)$ and $\texttt{cos}(x)$. Our method is then extended to $\texttt{tan}(x)$.

**Inverse trigonometric functions:** In the same direction as our other contributions in this paper, we propose an approach to build oblivious inverse trigonometric functions, by using numerical approximations. We again use the methods of Hart here. Bayatbabolghani et al. [4], introduced a protocol based on the work of Medina [24], using a sequence of polynomials to obtain the `arctan`$(\langle x\rangle)$ of a secret shared input $x$. In their work, they provided secure protocols to achieve this, and some discussion about complexity and performance, comparing it with [16]. However no specific implementation details were provided, except for the usage of these operations within spectrum fingerprint detection algorithms. Their experimentation was performed using the PICCO compiler [27].

**The functions exp and log:** Following the same methodology, we present protocols for algorithms such as exponentiation (considering the base and exponent as secret shared inputs) and log (to a public available base). We achieve this, by numerically approximatingtoWe both operations using the methods of Hart [16] (base two), and then making use of standard logarithmic identities, with the aim of computing both functions. Previous work on exponentiation algorithms, initially used binary expansions and utilized existing work on bit-decomposition for field elements. The basic relevant work in this regard was introduced by Damgård et al. [11], where the authors hide both the exponent and the base by evaluating the binary expansion of the exponent. Further works have centered on reducing the influence of the binary expansion [25,26]. For logarithmic functions, there is virtually no prior work on fixed point MPC variants, although there has been some work on floating point variants. For instance, Kamm [17] makes use of Taylor series to obtain the natural logarithm of a given floating point input.

In summary we provide methods to securely compute various scientific operations, and we evaluate their performance in practice using an off-the-shelf MPC system. We hope our work stimulates others to investigate improvements to our methods. Our choice of fixed point representation is to enable fast secure evaluation of the scientific functions, clearly it would be better to use floating point representations.

## 2 Preliminaries

In this section we outline the necessary details to understand the following contributions. In particular, how we perform fixed point arithmetic, numerical approximation, our arithmetic black box, as well as our experimental setup.

**Notation for fixed point arithmetic:** We make use of the square bracket notation from [13], where $[\![a]\!]$ denotes a secret shared value $a \in \mathbb{F}_q$. Note that our protocols are designed to work regardless of the underlying Linear Secret Sharing Scheme (LSSS). We assume all our inputs are elements of some field elements $\mathbb{F}_q$, where $q$ is a prime of bit-size $\ell$. We use typical assumptions while encoding integer values in $\mathbb{F}_q$. That is to say that we consider half of the input domain to represent positive numbers, and the other half negative. Let $P$ be the set of all parties of size $|P|$.

We follow a representation proposed by Catrina and Saxena [8], which is common in the MPC literature and libraries [2–4]. We define $\mathbb{Z}_{\langle k \rangle}$ as the set of integers $\{x \in \mathbb{Z} : -2^{k-1} \leq x \leq 2^{k-1} - 1\}$, which we embed into $\mathbb{F}_q$ via the map $x \mapsto x \pmod{q}$. We define $\mathbb{Q}_{\langle k,f \rangle}$ as the set of rational numbers $\{x \in \mathbb{Q} : x = \overline{x} \cdot 2^{-f}, \overline{x} \in \mathbb{Z}_{\langle k \rangle}\}$. We represent $x \in \mathbb{Q}$ as the integer $x \cdot 2^f = \overline{x} \in \mathbb{Z}_{\langle k \rangle}$, which is then represented in $\mathbb{F}_q$ via the mapping used above. Thus $x \in \mathbb{Q}$ is in the range $[-2^e, 2^e - 2^{-f}]$ where $e = k - f$. As we are working with fixed point numbers we assume that the parameters $f$ and $k$ are public. For our following algorithms to work (in particular fixed point multiplication and division) we require that $q > 2^{2 \cdot k}$. We can then imagine a minimal representation of a secret shared fixed point number $x$, as $\langle x \rangle$ to be a tuple composed by $\{[\![v]\!], k, f\}$. We extend the notation in [13], encoding secret shared field elements as $[\![x]\!]$ and fixed-point inputs as $\langle x \rangle$. Note that operations with public fixed-point operations are possible by using the same basic encoding. Vectors of secret shared inputs are also denoted by $[\![Y]\!]$ or $\langle Y \rangle$, and its size is $|Y|$, with the context being implicitly clear.

**Experimental setup:** All experiments in this paper were run using a LAN network test-bed (10 Gb switch and connections), with dedicated machines. Each machine had the same hardware and software configuration, namely 32 GB RAM, 256 SSD storage, Intel Core i7-770 3.6GHZ processor, and were running Ubuntu 16.04.5 LTS. The machines ran the `SCALE-MAMBA` system [2] for their base MPC protocols. This is an MPC framework which runs in the offline-online paradigm, namely work is performed in two distinct phases: a function independent offline phase (used to generate correlated randomness) and an online phase (where the function is evaluated). More specifically, the former phase is dedicated to generating Beaver Triples [5] and random shared bits.

`SCALE-MAMBA` allows us to test our protocols in an actively secure environment (in particular active security with abort) for various access structures. For comparison purposes we looked at three setups;

- A two and three party full-threshold access structure which uses (essentially) the SPDZ protocol from [12, 14, 19],
- A three party honest majority setting using Shamir secret sharing. This variant uses Maurer's protocol [23] to generate offline data, which is then processed as in [20].

`SCALE-MAMBA` has built in protocols for performing fixed point arithmetic based on the methodology of [8] described above. We made use of the default configuration of `SCALE-MAMBA` to run all our experiments, except for those for the exponentiation and logarithm functions. This implies that we used a 128-bit modulus for $\mathbb{F}_q$. Additionally, fixed point inputs are $k = 41$ bits in length of which $f = 20$ bits are dedicated to its fixed point precision. This implies an implicit statistical security parameter $\kappa$ for the fixed point arithmetic emulation of 40 bits.

We note that the implementation of `SCALE-MAMBA` optimizes execution times by running parallel threads to create offline data "just-in-time". However, for the cases where the offline phase can be executed in advance, we also run experiments to measure exclusively the online phase and to estimate the execution time of any associated offline phase. Communication cost greatly influences the overall running times of the system; and because of this the compiler will try to optimize execution times so as to maximize throughput. This is done by executing multiple operations in a single round. Our experiments include configurations for cases when compilation is optimized in this way, and when is not. Thus, we get estimates for when one wants to maximize throughput, and when one wants to minimize latency.

In all our experimental reports in what follows we present three figures.

- **Offline Phase**. We measure the average time it takes to produce enough triples during the offline phase for a single execution of the functionality (e.g., a single $\langle \mathtt{sin}(x) \rangle$ or $\langle \mathtt{tan}(x) \rangle$ call).
- **Latency Measurement**. In this setting we evaluate the online phase of our protocol executing a single operation at a time (i.e. sequential as opposed to parallel execution). We then present the average run time for the online phase only. This gives an estimate of the expected latency a user can expect if latency of computation is the main performance issue.
- **Throughput Maximization**. We also measured when we run several instances of the functionality (in our case 50) in parallel. Thus this enables us to give a *lower bound* for the expected throughput, i.e. how many operations can be performed per second, if throughput is the main performance issue.

Note that computational costs for the offline phase dominate on overall performance, and that in `SCALE-MAMBA`, the offline phase works on the same way regardless of whether the online phase is configured to maximise throughput or minimize latency.

**Arithmetic Black Box:** To facilitate the understanding of the implications of using functionalities that are as secure as the underlying MPC protocols that implement them, we follow literature on the field by describing an *arithmetic blackbox ($\mathcal{F}_{ABB}$)*. This was originally introduced by [13], in the context of abstracting away finite field operations in $\mathbb{F}_q$ via shares $[\![x]\!]$, but one can also extend it to operations on our fixed point sharing $\langle x \rangle$, as well as more complex operations which have already been proved to be secure under composition.

The $\mathcal{F}_{ABB}$ works as an idealized functionality, capable to store secret values over $\mathbb{F}_q$ (input) and make them public under request (output). A stored $x \in \mathbb{F}_q$ will be denoted by $[\![x]\!]$. Furthermore, it can perform a series of operations under request by the computational parties, for example addition and multiplication of of such elements. Hence, it can be asked to compute any function, by constructing the associated functionality as an arithmetic circuit. This allows our protocols to abstract themselves from the specific details of how the MPC system implements them. The basic functionality, which includes addition and multiplication of field elements as well as fixed-point inputs and is detailed in Table 1. With the protocols used to implement these functions, in our experiments, being taken from the underlying protocols in `SCALE-MAMBA` described above. In the same table, we also present the number of rounds needed to execute each function in the online phase of the `SCALE-MAMBA` system. Additionally, we make occasional use of high level functionalities which have been given and proven secure by various other authors. These protocols are given by Table 2.

| **Operation** | **Purpose** | **Rounds** |
|---|---|---|
| $x \leftarrow \llbracket x \rrbracket$ | Opening/outputing a secret field element | 1 |
| $\llbracket x \rrbracket \leftarrow x$ | Inputing secret a field element | 1 |
| $\llbracket z \rrbracket \leftarrow \llbracket x \rrbracket + \llbracket y \rrbracket$ | Adds secret field elements | 0 |
| $\llbracket z \rrbracket \leftarrow \llbracket x \rrbracket + y$ | Adds secret field and public element | 0 |
| $\llbracket z \rrbracket \leftarrow \llbracket x \rrbracket \cdot y$ | Multiplies secret field and public element | 0 |
| $\llbracket z \rrbracket \leftarrow \llbracket x \rrbracket \cdot \llbracket y \rrbracket$ | Multiply secret elements | 1 |
| $\langle z \rangle \leftarrow \langle x \rangle + \langle y \rangle$ | Adds secret fixed point numbers | 0 |
| $\langle z \rangle \leftarrow \langle x \rangle + y$ | Adds secret and public fixed point numbers | 0 |
| $\langle z \rangle \leftarrow \langle x \rangle \cdot \langle y \rangle$ | Multiplies secret fixed point numbers | 1 |
| $\langle z \rangle \leftarrow \langle x \rangle \cdot y$ | Multiplies secret and public fixed point numbers | 0 |
| $\langle z \rangle \leftarrow \llbracket x \rrbracket + \langle y \rangle$ | Adds secret fixed point number with secret field element | 0 |
| $\langle z \rangle \leftarrow \llbracket x \rrbracket \cdot \langle y \rangle$ | Multiplies secret fixed point number with secret field element | 1 |

**Table 1.** Secure Arithmetic operations provided by the $\mathcal{F}_{ABB}$.

| **Operation** | **Purpose** | **Rounds** | **Protocol** |
|---|---|---|---|
| $\llbracket b \rrbracket \leftarrow \llbracket x \rrbracket < \llbracket y \rrbracket$ | Compares a secret and field elements | $1 + \texttt{log}_2(\ell)$ | [7] |
| $\llbracket b \rrbracket \leftarrow \llbracket x \rrbracket < y$ | Compares a secret and public field elements | $1 + \texttt{log}_2(\ell)$ | [7] |
| $\llbracket b \rrbracket \leftarrow \langle x \rangle < \langle y \rangle$ | Compares a secret and fixed point numbers | $1 + \texttt{log}_2(\ell)$ | [7] |
| $\llbracket b \rrbracket \leftarrow \langle x \rangle < y$ | Compares a secret and public fixed point numbers | $1 + \texttt{log}_2(\ell)$ | [7] |
| $\langle z \rangle \leftarrow \langle x \rangle / \langle y \rangle$ | Divides secret fixed point numbers | $2 \cdot \log_2(\frac{k}{3.5}) + 8$ | [8] |
| $\langle z \rangle \leftarrow \texttt{choose}(\llbracket b \rrbracket, \langle x \rangle, \langle y \rangle)$ | `MUX`. Returns $\langle x \rangle$ or $\langle y \rangle$ depending on bit $\llbracket b \rrbracket$ s.t. $(\langle y \rangle - \langle x \rangle) \cdot \llbracket b \rrbracket + \langle x \rangle$ | 1 | [11] |
| $\langle z \rangle \leftarrow \texttt{choose}(b, \langle x \rangle, \langle y \rangle)$ | `MUX`. Returns $\langle x \rangle$ or $\langle y \rangle$ depending on bit $b$ s.t. $(\langle y \rangle - \langle x \rangle) \cdot b + \langle x \rangle$ | 1 | [11] |
| $\llbracket b \rrbracket_0, ..., \llbracket b \rrbracket_{\ell-1} \leftarrow \texttt{bit\_decompose}(\llbracket x \rrbracket)$ | Bit decomposition of secret field element | $\log_2(q)$ | [11, 26] |
| $\llbracket b \rrbracket_0, ..., \llbracket b \rrbracket_{k-1} \leftarrow \texttt{pre\_OR}(\llbracket x \rrbracket_0, .., \llbracket x \rrbracket_{k-1})$ | fan-in `or` | 1 | [11] |
| $\llbracket z \rrbracket \leftarrow \texttt{trunc}(\langle x \rangle)$ | $\texttt{trunc}(x)$ so that returns $x$'s integral magnitude | 2 | [8] |

**Table 2.** Secure complex functionalities derived from the $\mathcal{F}_{ABB}$.

*On mixed type operations:* In some cases, and as denoted by Table 1, we may need to either add or multiply secret share fixed point with a standard shared modulo $p$ value. In this case we are assumed to know a bound on the number of bits in the shared modulo $p$ value; i.e. it is never a general element in $\mathbb{F}_p$ but one of bounded size when reduced to a centre around zero. As described by Catrina and Saxena [8] we need to scale the integer inputs by $2^f$. This way, integer operands share the same encoding than their fixed point counterparts. This process is called "scaling", and can be achieved by shifting the input to the left by $|f|$ bits. We refer the reader to [8], for a more complete explanation of this process.

Note that subtractions, can be trivially derived from the functionality described by Table 1. Given that we can encode negative numbers, subtractions can be seen as a special case of addition, where the substracted input is multiplied by $-1$.

**On numerical approximation:** As it was previously mentioned, our protocols are based on the results outlined by Hart in his work, *Computer Approximations* [16]. We make use of numerical methods that, through the use of Polynomial and Padé approximants, can obtain "good enough" approximations to transcendental functions, over a given input interval. To be able to operate, we reduce (normalize) our inputs to such intervals and keep track of their cyclic position. Throughout this work, polynomials are referred in the same way as in the original work by Hart; that is to say a polynomial is described as capital $P_i$ and $Q_i$ where $i$, refers to the index of table, taken directly from Hart's work [16]. In the case of a polynomial approximation to a function $f$ we have $f(x) \approx P_i(x)$, whilst in the case of a Padé approximation we have $f(x) \approx P_i(x)/Q_i(x)$. We have included, an appendix with all the polynomials used in this work, as well as their precision.

## 3 Approximated Square Root

In this section, we introduce our results, with respect to the oblivious computation of the square root. The intuition of our work is as follows, given a shared fixed point number $\langle x \rangle = \{[\![v]\!], k, f\}$; we create an initial approximation of the form $\langle v \rangle / 2^{\frac{[\![m]\!]}{2}}$, where $[\![m]\!]$ is the secret shared location of the most significant bit in $[\![v]\!]$. Following Liedel [22], we then improve our approximation by using a number of Newton-Raphson and Goldschmidt iterations. However, the initial approximation to the square root presented by Liedel does not work on all possible input numbers. In particular the approximation algorithm requires a fixed point division by the number $2^{3 \cdot k - 2 \cdot f} = 2^t$. To perform this we create the clear fixed point representation of the value $1/2^t$ and then perform a multiplication between a clear and a shared value. However, to represent $1/2^t$ in our fixed point representation we require there to be a value $i \in [0, \dots, k)$ such that $i - f = -t$, i.e. $t = f - i < f$. Thus there are some inputs for which Liedel's method to produce the first approximation would require us to increase our precision, and hence our costs, and potentially the underlying prime size. However, we will see that a more crude initial approximation suffices.

**Most Significative Bit:** Our protocols require us to identify the Most Significative Bit (MSB) from any $\mathbb{F}_q$ element. To achieve this, we adapt the results from [11, 22], in such a way that we can isolate it. Our adapted construction makes use of the following inputs:

- $\llbracket \mathbf{v} \rrbracket$: Integer input value.
- $k$: Represents a bound on the size of $v$ as an integer. In particular $|v| < 2^k$.

Our protocol will return the most significative bit, which is less than $k$, but encoded as an index vector. Protocol 1, encompass the method used to achieve this. Note that, to improve the understanding (implementation-wise) of the protocols in this section (and, in particular Protocol 2), the output index vector will be of size $k$ when $k$ is even and $k+1$ when $k$ is odd. This is needed to enable the indexing in our parity extraction step in Protocol 2 to be correct.

**Protocol 1:** Most Significative Bit Extraction

```
Input: Secret shared integer input$\llbracket v \rrbracket$. Bit-wise upper bound $k$
Output: Returns secret shared vector $\llbracket z \rrbracket$ with $z \in \{0,1\}^k$ or $\{0,1\}^{k+1}$, which
        is all zero except for the location of the MSB of $v$.
$\llbracket V \rrbracket_b \leftarrow$ bit_decompose($\llbracket v \rrbracket$);
$\llbracket V' \rrbracket_b \leftarrow \{0_1, ..., 0_{|V_b|}\}$;
for $i \leftarrow k$ to 1 do
    $\llbracket V' \rrbracket_{l-i+1} \leftarrow \llbracket V \rrbracket_i$;    //invert its order
$\llbracket Y \rrbracket \leftarrow$ pre_OR($\llbracket V' \rrbracket_b$);
$\llbracket Y' \rrbracket \leftarrow \{0_1, ..., 0_{|Y|}\}$;
for $i \leftarrow k$ to 1 do
    $\llbracket Y' \rrbracket_{k-i+1} \leftarrow \llbracket Y \rrbracket_i$;    //restore its order
$\llbracket z \rrbracket \leftarrow \{0_1, ..., 0_{k+1-(k \mod 2)}\}$;
for $i \leftarrow 1$ to $k-1$ do
    $\llbracket z \rrbracket_i \leftarrow \llbracket Y' \rrbracket_i - \llbracket Y \rrbracket_{i+1}$;
$\llbracket z \rrbracket_k \leftarrow \llbracket Y' \rrbracket_k$;
return $\llbracket z \rrbracket$;
```

The protocol works by first obtaining the bit decomposition of our field element, we then obtain the fan-in OR (`pre_OR`($\llbracket x \rrbracket$)) of the binary expansion of the input, in inverse order. Note that, without loss of generality, our protocol is explained by using full integers of size $k$, however it can be used to select the MSB of any substring of size smaller than $k$. We then simply proceed to obliviously identify the point where, the `pre_OR`($\llbracket a \rrbracket$) stop returning $\llbracket 0 \rrbracket$ and become $\llbracket 1 \rrbracket$. Finally, we adjust the return vector size depending on $k$.

**Initial Square Root Approximation:** By extracting the MSB of the input we can obtain our initial approximation via $[\![w]\!] \leftarrow 2^{\frac{[\![m]\!]}{2}}$ if $[\![m]\!]$ is even, or $\langle w \rangle \leftarrow 2^{\frac{[\![m-1]\!]}{2}}$ if odd, where $[\![m]\!]$ is the MSB of $[\![v]\!]$. Note, this needs to be done without disclosing the parity of $m$, as explained in Protocol 2. This would suffice to obtain the desired approximation. Additionally, we have to deal with the effects of the parity of $f$ (by making minor changes depending on whether $f$ is even or odd).

**Protocol 2:** Approximation of the Square Root (`app_sq`)

```
Input: Secret shared integer input $[\![v]\!]$, and bit-wise upper bound $k$.
Output: MSB index position $[\![m]\!]$, its parity $[\![o]\!]$ and a power of 2
        approximation for $\langle \sqrt{x} \rangle$ in $[\![w]\!]$
$[\![z]\!] \leftarrow$ MSB($[\![v]\!], k$); (i.e. Protocol 1)
$[\![m]\!] \leftarrow [\![0]\!]$;
$[\![o]\!] \leftarrow [\![0]\!]$;   //is odd
for $i \leftarrow 1$ to $k$ do
    $[\![m]\!] \leftarrow [\![m]\!] + (i) \cdot [\![z]\!]_{i-1}$;
    if $(i \mod 2) = 1$ then
        $[\![o]\!] \leftarrow [\![o]\!] + [\![z]\!]_i$;
$[\![W]\!] \leftarrow \{[\![0]\!]_1, ..., [\![0]\!]_{\lceil \frac{k}{2}+1 \rceil}\}$;   //size is $\lceil \frac{k}{2} \rceil + 1$
$[\![W]\!] \leftarrow [\![0]\!]$;
for $i \leftarrow 1$ to $\frac{k}{2} + 1$ do
    $[\![w]\!] \leftarrow [\![w]\!] + (2^{i-1}) \cdot [\![W]\!]_i$;
return $[\![o]\!], [\![m]\!], [\![w]\!]$;
```

The protocol converts the sharing of the $\{0,1\}^k$ vector in $[\![z]\!]$ produced by Protocol 1 into an integer sharing $[\![m]\!]$ with $m \in \{1, ..., k\}$. At the same time we identify the parity of $m$, and we then calculate $[\![w]\!]$ by evaluating the binary expansion of the index encoding vector.

**Privacy Preserving Square Root:** Once, we have obtained our initial approximation, following Liedel [22] results, which we present in Protocol 3. We fix the maximum number of iterations for the Goldshmidt Newton-Raphson combination, just as in Liedel [22], and assign it to $\theta$. The precision of our construction is tied to $\theta$ and, has to be tunned in according to the application at hand. Our experiments yielded an accuracy of around six digits after six repetitions. We first obtain $[\![w]\!]$ by invoking Protocol 2 and then proceed to build an instance of $[\![w]\!]$ as a fixed point number, in accordance to the parity of $f$, $[\![m]\!]$ and Protocol 2. The protocol works by executing a Goldschmidt's iteration followed by a final Newton iteration.

**Results:** To provide a comparison with the previous work of Liedel, we also present run-times for his results as well. However, we stress again that Liedel's

**Protocol 3:** Optimized Approximated $\langle\sqrt{x}\rangle$ for fixed point

```
Input: Secret shared fixed point input $\langle x\rangle = \{[\![v]\!], k, f\}$.
Output: Square root of input $\langle\sqrt{x}\rangle$.
$\theta \leftarrow \mathtt{max}(\lceil\log_2(x_k)\rceil, 6)$;
$[\![o]\!], [\![m]\!], [\![w]\!] \leftarrow \mathtt{app\_sq}([\![x_v]\!], x_k)$;
$[\![o]\!] \leftarrow \mathtt{choose}((f \mod 2), 1 - [\![o]\!], [\![o]\!])$;
$[\![w]\!] \leftarrow \mathtt{choose}((1 - [\![o]\!]) \cdot (f \mod 2), 2 \cdot [\![w]\!], [\![w]\!])$;
$t \leftarrow (f - (f \mod 2))/2$;
$\langle w'\rangle \leftarrow ([\![w]\!] \cdot 2^t, k - f, f)$;
$\langle w'\rangle \leftarrow \mathtt{choose}([\![o]\!], \langle w'\rangle \cdot \sqrt{2}, \langle w'\rangle)$;
$\langle y\rangle \leftarrow \langle w'\rangle^{-1}$;
$\langle g\rangle \leftarrow \langle x\rangle \cdot \langle y\rangle$;
$\langle h\rangle \leftarrow \langle y\rangle/2$;
for $i \leftarrow 1$ to $\theta$ do
    $\langle r\rangle \leftarrow \frac{3}{2} - \langle g \cdot h\rangle$;
    $\langle g\rangle \leftarrow \langle g\rangle \cdot \langle r\rangle$;
    $\langle h\rangle \leftarrow \langle h\rangle \cdot \langle r\rangle$;
$\langle r\rangle \leftarrow \frac{3}{2} - \langle g \cdot h\rangle$;
$\langle h\rangle \leftarrow \langle h\rangle \cdot \langle r\rangle$;
$\langle H\rangle \leftarrow 3 - 4 \cdot \langle x\rangle \cdot \langle h\rangle^2$;
$\langle H\rangle \leftarrow \langle h\rangle \cdot \langle H\rangle$;
$\langle\sqrt{x}\rangle \leftarrow \langle x\rangle \cdot \langle H\rangle$;
```

method is not as general as the method we propose, as we can cope with a much larger set of input parameters. In Table 3 we present the required offline data, per single square root operation. Then in Table 4 we present the execution times for the offline phase (for a single square root operation), plus the minimum latency and maximum throughput we obtained in the three different configurations we tested. We see that, when Liedel's method can be applied then the performance is better, but the extra cost of our method in dealing with general inputs is only about a factor of two.

| Protocol | Liedel | This Work |
|---|---|---|
| Multiplication Triples | 197 | 684 |
| Square Tuples | 1 | 0 |
| Shared Bits | 2598 | 4049 |

**Table 3.** Offline data needed for a single fixed point square root operation

| | Full Threshold 2 Parties | | Full Threshold 3 Parties | | Shamir 3 Parties | |
|---|---|---|---|---|---|---|
| Protocol | Liedel | This Work | Liedel | This Work | Liedel | This Work |
| Offline (sec) | 2.49 | 3.90 | 2.905 | 4.708 | 0.065 | 0.088 |
| Latency (sec) | 0.0034 | 0.0048 | 0.0043 | 0.0060 | 0.0039 | 0.0062 |
| Throughput (ops/sec) | 1042 | 491 | 795 | 313 | 785 | 308 |

**Table 4.** Performance figures for Full Threshold (2 and 3 parties) and Shamir with 3 parties for the fixed point square root calculation.

**Discussion:** A recent implementation, with regards to the secure evaluation of square root functions on distributed environments, was introduced by Dimitrov et al. [15]. These implementations where part of their work on alternative representations for real numbers. Their results make use of a golden number encoding, as well as some logarithmic based representation for real numbers. The authors followed the Liedel line of work for their implementation, and used (the passively secure) Sharemind [6] system as their test-bed. They made use of a similar configuration (3 parties) on high-end machines, except for the fact they ran their experiments using Intel Xeon microprocessor series. For comparison reasons, they included experimentation against 32 and 64 bit long fixed point representations included in Sharemind. Their work however only gives estimations on the number of operations per second, but no mention on whether these are batched together. The fastest implementation is their logarithmic representation using a low bit-size for the inputs and a somewhat small precision. Direct comparison is hard to make as they target only passively secure MPC, whereas we focus on actively secure MPC.

## 4 Trigonometric Functions

We introduce a series of adaptations of the numerical approximations given by Hart [16] for the basic trigonometric functions, and then implement them in an oblivious fashion. The approximations have been chosen to balance accuracy and low degree (i.e. efficiency).

**Angle reduction:** We first introduce a mechanism, Protocol 4, to map any input $\langle x\rangle$ to the range $[0, \frac{\pi}{2}]$, and the quadrant which $\langle x\rangle$ lies in (given by $b_1$ and $b_2$). The quadrant is a byproduct of the process of the initial mapping. Protocol 4 requires a `trunc`$(x)$ operation call, and a low number of fundamental operations. The outputs of the operation, are as follows:

- $\langle w\rangle$: $w = x \pmod{\pi/2}$.
- $[\![b_1]\!]$: $b_1 = (x \pmod{2\cdot\pi}) \;>\; \pi$.
- $[\![b_2]\!]$: $b_2 = (x \pmod{\pi}) \;>\; \pi/2$.

**Sine, Cosine and Tangent Functions:** First, the input $\langle x\rangle$ has to be mapped to the correct interval. We then can obtain the sin of any angle by using the polynomial approximation $\sin(x) = \nu \cdot P_{3307}(\nu^2)$ where $\nu = w\cdot 2/\pi$ for $w = x$ (mod $\pi/2$), with the polynomial $P_{3307}$ from Hart [16] given in the Appendix. We can produce the cosine function, by evaluating $P_{3508}(w^2)$ from [16] (again details in Appendix). Given the cyclic nature of both sin and cos, we adjust the sign of the outputs by $b_1$ and $b_2$ accordingly. Finally, using the standard identity, $\tan(x) = \sin(x)/\cos(x)$ we can then give the tangent function.

**Protocol 4:** Angle reduction protocol

```
Input: Secret shared fixed point input $\langle x\rangle = \{[\![v]\!], p, f\}$.
Output: Secret shared reduced angle $\langle w\rangle$ such that $0 \leq \langle w\rangle \leq \frac{\pi}{2}$, and flags $[\![b_1]\!]$
        and $[\![b_2]\!]$.
$\langle d\rangle \leftarrow \langle x\rangle \cdot \frac{1}{2\cdot\pi}$; //This is a scalar mult.
$[\![d]\!] \leftarrow \texttt{trunc}(\langle d\rangle)$;
$\langle y\rangle \leftarrow \langle x\rangle - [\![d]\!] \cdot (2\cdot\pi)$;
$[\![b_1]\!] \leftarrow \langle y\rangle > \pi$;
$\langle w\rangle \leftarrow \texttt{choose}([b_1], (2\cdot\pi) - \langle y\rangle, \langle y\rangle)$;
$[\![b_2]\!] \leftarrow \langle w\rangle > \frac{\pi}{2}$;
$\langle w\rangle \leftarrow \texttt{choose}([\![b_2]\!], (\pi - w) - \langle w\rangle, \langle w\rangle)$;
return $\langle w\rangle, [\![b_1]\!], [\![b_2]\!]$;
```

**Inverse trigonometric functions** Inverse trigonometric functions can be built directly, from an approximation to the arctan function via

$$\texttt{arcsin}(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right) \text{ and } \texttt{arccos}(x) = \frac{\pi}{2} - \texttt{arcsin}(x).$$

For arctan we have to perform a somewhat similar input reduction procedure as was done for the main trigonometric functions above; as is also the case in [4]. We first simplify the process by operating on positive values only, since $\arctan(-x) = -\arctan(x)$. Thus, we first need to identify the sign of $\langle x\rangle$ sign. We then can reduce $\langle x\rangle$ value to the interval $[0, 1]$, by using the formula $\arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right)$. From this point, it suffices to obtain an approximation for $\texttt{arctan}(x)$ in the interval $x \in [0, 1]$. Which we again do via a Padé approximation $P_{5102}(X)/Q_{5102}(X)$ from [16] (see the Appendix). Protocol 5 shows how we obtain this value, by using the building blocks, enumerated in previous sections.

**Protocol 5:** Approximated $\texttt{arctan}(\langle x\rangle)$

```
Input: Secret shared fixed point input $\langle x\rangle = \{[\![v]\!], k, f\}$.
Output: Approximation for $\langle\texttt{arctan}(\langle x\rangle)\rangle$
$[\![s]\!] \leftarrow \langle x\rangle < 0$;
$\langle \texttt{abs}(x)\rangle \leftarrow \texttt{choose}([\![s]\!], -1\cdot\langle x\rangle, \langle x\rangle)$;
$[\![b]\!] \leftarrow \langle \texttt{abs}(x)\rangle > 1$;
$\langle \nu\rangle \leftarrow \texttt{choose}([\![b]\!], \frac{1}{\langle \texttt{abs}(x)\rangle}, \langle \texttt{abs}(x)\rangle)$;
$\langle y\rangle \leftarrow P_{5102}(\langle \nu\rangle^2)/Q_{5102}(\langle \nu\rangle^2)$;
$\langle \texttt{arctan}(\langle x\rangle)\rangle \leftarrow \langle x\rangle \cdot \langle y\rangle$;
```

**Results:** Just as in the case of the square root function we present our results in two tables. To calculate $\langle\texttt{arccos}(x)\rangle$, trivially follows from solving $\langle\texttt{arcsin}(x)\rangle$, hence, running times are essentially the same and thus ignored. The first table, Table 5, gives the offline cost per function call, whereas the second, Table 6, gives the actual measured costs using our programs.

| Protocol | ⟨sin()⟩ | ⟨cos()⟩ | ⟨tan()⟩ |
|---|---|---|---|
| Multiplication Triples | 267 | 266 | 680 |
| Square Tuples | 1 | 1 | 1 |
| Shared Bits | 3806 | 3562 | 8018 |

| Protocol | ⟨arcsin()⟩ | ⟨arctan()⟩ |
|---|---|---|
| Multiplication Triples | 2053 | 967 |
| Square Tuples | 0 | 0 |
| Shared Bits | 13431 | 7732 |

**Table 5.** Offline data needed for a single fixed point trigonometric operation

| | Full Threshold<br>2 Parties | | | Full Threshold<br>3 Parties | | | Shamir<br>3 Parties | | |
|---|---|---|---|---|---|---|---|---|---|
| Protocol | ⟨sin()⟩ | ⟨cos()⟩ | ⟨tan()⟩ | ⟨sin()⟩ | ⟨cos()⟩ | ⟨tan()⟩ | ⟨sin()⟩ | ⟨cos()⟩ | ⟨tan()⟩ |
| Offline (sec) | 3.78 | 3.37 | 7.89 | 4.62 | 4.24 | 9.49 | 0.084 | 0.084 | 0.18 |
| Latency (sec) | 0.0042 | 0.0035 | 0.0053 | 0.0050 | 0.0043 | 0.0076 | 0.0044 | 0.0045 | 0.0078 |
| Throughput (ops/sec) | 617 | 561 | 289 | 542 | 724 | 301 | 446 | 467 | 239 |

| | Full Threshold<br>2 Parties | | Full Threshold<br>3 Parties | | Shamir<br>3 Parties | |
|---|---|---|---|---|---|---|
| Protocol | ⟨arcsin()⟩ | ⟨arctan()⟩ | ⟨arcsin()⟩ | ⟨arctan()⟩ | ⟨arcsin()⟩ | ⟨arctan()⟩ |
| Offline (sec) | 15.43 | 7.36 | 18.3 | 8.70 | 0.56 | 0.18 |
| Latency (sec) | 0.024 | 0.0068 | 0.028 | 0.0082 | 0.030 | 0.0089 |
| Throughput (ops/sec) | 41 | 275 | 57 | 218 | 40 | 191 |

**Table 6.** Performance figures for Full Threshold (2 and 3 parties) and Shamir with 3 parties for the fixed point trigonometric operations.

**Discussion:** There are some related works that explore similar results but differ in regards to the underlying method to compute on encrypted data. The problem of computing trigonometric functions using Homomorphic Encryption was most recently addressed by Cheon et al. [9]. In their work they tackle various topics related to fixed point representation in the homomorphic encryption domain. The authors included results for several operations related to statistical functions. Amongst them, the authors provided timings for sigmoids, using Taylor series. Their test-bed (a single machine) was fairly similar to the set-up of our own machines, but they have used Intel i5 processors instead. On their timings themselves, they were slightly slower than what it was achieved by this work. Namely, their variant of the `HEEAN` protocol, was capable of evaluating sigmoid functions in around 167 ms (non-amortized cost), whereas our most common set-up, which considers three parties using Shamir's secret sharing such a function could be evaluated in 4 ms. As a final note, it has to be taken into account that Homomorphic Encryption and MPC are often targetted at different scenarios, thus making difficult to establish direct comparison, given that the protocol selection does not exclusively depends on their performance.

## 5 Exponentiation and Logarithms

In this section, we explore how to obtain the power and the logarithm of any base, to any exponent. This can be achieved by the use of standard logarithmic identities and numerical approximations; namely $\texttt{log}_b(x) = \texttt{log}_2(b) \cdot \texttt{log}_2(x)$ and $\texttt{exp}(x, y) = x^y = \texttt{exp}(2, y \cdot \texttt{log}_2(x))$.

**Logarithmic Function:** To calculate $\langle \texttt{log}_2(x) \rangle$, we first need to express $\langle x \rangle$ using the secret shared floating point notation used in [1]. This is to enable us to extract the normalized value of $x$ in the range $[0.5, 1]$, to enable the calculation of the function via numerical approximation. We denote this operation as follows:

$$(\llbracket v_f \rrbracket, \llbracket f_f \rrbracket, \llbracket s \rrbracket, \llbracket z \rrbracket) \leftarrow \texttt{f\_cast}(\langle x \rangle),$$

which produces the elements to encode the shared fixed point number $\langle x \rangle$ as a shared floating point number. The details of these elements is as follows:

- $\llbracket s \rrbracket$ is a sharing of the sign of $x$.
- $\llbracket z \rrbracket$ is a sharing determining whether $x$ is zero or not.
- $\llbracket f_f \rrbracket$ is the secret shared significand for the representation.
- $\llbracket v_f \rrbracket$ is the mantissa, namely an integer value which is normalized to be in the range $[2^{k-1}, \ldots, 2^k)$.

The underlying floating point number can thus be expressed as $(1 - 2 \cdot s) \cdot (1 - z) \cdot v_f \cdot 2^{f_f}$. To compute $\texttt{f\_cast}(\langle x \rangle)$ we make use of the method introduced by Aliasgari et al. [1]. Internally, this functionality determines the position of the MSB in $\llbracket v \rrbracket$, this enables us to obtain the number of bit shifts needed to compute $\llbracket v_f \rrbracket$ and, hence, $\langle f_f \rangle$ from the $\llbracket v \rrbracket$ and $f$ values used to represent $\langle x \rangle$. We direct the reader to [1] for a more complete explanation of this conversion routine.

Let us define $\llbracket e_f \rrbracket = k + \llbracket f_f \rrbracket$. To obtain the $\langle \texttt{log}_2(x) \rangle$ we map $\langle x \rangle$ to the range $[0.5, 1]$, by computing $\langle \nu \rangle = \langle \frac{1}{2^k} \rangle \cdot \llbracket v_f \rrbracket$. Then we can use it to compute $\langle \texttt{log}_2(x) \rangle = \llbracket e_f \rrbracket + \langle \texttt{log}_2(\nu) \rangle$. The approximation to $\langle \texttt{log}_2(\nu) \rangle$ can then be produced by a Padé approximation, calculated by the means of the $P_{2524}/Q_{2524}$ polynomials, introduced by [16] (and given in the Appendix). Note that we define for this function $\log_2(x) = 0$ when $x \leq 0$. The motivation behind this behaviour is given because, including any abort would signal, when the answer is opened, information related to the input.

**Exponentiation Functions:** We are left with deriving $\langle \texttt{exp}(2, x) \rangle$, for a secret shared input $\langle x \rangle$. Due to standard identities, this can be obtained from a polynomial approximation to $\texttt{exp}(2, x)$ in the interval $[0, 1]$. In this regard, we first need to isolate the integral part $\llbracket i \rrbracket$ and fractional remainder $\langle r \rangle$ of the input value $\langle x \rangle$ such that $\langle x \rangle = \llbracket i \rrbracket + \langle r \rangle$. We can then calculate $2^{\llbracket i \rrbracket}$, using conventional techniques for bit-decomposition and exponentiation e.g., [11]. We can obtain $2^{\langle r \rangle}$ using a polynomial approximation, by means of $P_{1045}(\langle r \rangle)$, as given in the Appendix. From that point on, it suffices to follow the identities outlined at the beginning of this section to obtain $\langle \texttt{exp}(x, y) \rangle$.

**Protocol 6:** Approximated $\langle \texttt{exp}(2,x)\rangle$

```
Input: Secret shared fixed point input $\langle x\rangle = \{[\![v]\!], k, f\}$.
Output: Approximation for $\langle \exp(2,x)\rangle$
$[\![s]\!] \leftarrow \langle x\rangle < 0$;
$\langle x\rangle \leftarrow$ choose($[\![s]\!], -1 \cdot \langle x\rangle, \langle x\rangle$); //Convert input to positive number
$[\![i]\!] \leftarrow$ trunc($x$); //extract integer component of x
$\langle r\rangle \leftarrow \langle x\rangle - [\![i]\!]$; //Extract fractional component of x
$([\![i]\!]_0, ..., [\![i_{\ell-1}]\!]) \leftarrow$ bit_decompose($[\![i]\!]$);
$[\![d]\!] \leftarrow \prod_{j=0}^{\ell-1}([\![i_j]\!] \cdot 2^{2^j} + 1 - [\![i_j]\!])$;
$\langle u\rangle \leftarrow P_{1045}(\langle r\rangle)$;
$\langle g\rangle \leftarrow \langle u\rangle \cdot [\![d]\!]$;
$\langle \texttt{exp}(2,x)\rangle \leftarrow$ choose($1 - [\![s]\!], \langle g\rangle, \frac{1}{\langle g\rangle}$);
```

| Protocol | $\langle \texttt{log}_2(x)\rangle$ | $\langle \texttt{exp}(2,x)\rangle$ |
|---|---|---|
| Multiplication Triples | 1880 | 1337 |
| Square Tuples | 0 | 1 |
| Shared Bits | 5937 | 7688 |

**Table 7.** Offline data needed for a single fixed point exp/log operation

**Results:** Under default precision parameters of the SCALE-MAMBA system, and because of the size of the polynomials used for our approximations of both base two functions i.e., $\langle \texttt{log}_2(x)\rangle$ and $\langle \texttt{exp}(2,x)\rangle$, numerical results become less accurate and numerically unstable. Thus, to run our experiments in this example, we doubled the size of our inputs and their precision i.e. we use $k = 81$, $f = 40$, $\kappa = 80$. This, of course, influences the field size on which we operate, which has to be of at least 245 bits, instead of the 128 bits modulus used on our other experiments. Bigger field sizes also imply an increase on communication cost given that the size of the shares increases accordingly. Note that, as we use some level of bit decomposition in our protocols, the number of triples required also increases with the size of $k$ and $\kappa$. Our results are presented in Tables 7 and 8.

| | Full Threshold<br>2 Parties | | Full Threshold<br>3 Parties | | Shamir<br>3 Parties | |
|---|---|---|---|---|---|---|
| Protocol | $\langle \texttt{log}_2(x)\rangle$ | $\langle \texttt{exp}(2,x)\rangle$ | $\langle \texttt{log}_2(x)\rangle$ | $\langle \texttt{exp}(2,x)\rangle$ | $\langle \texttt{log}_2(x)\rangle$ | $\langle \texttt{exp}(2,x)\rangle$ |
| Offline (s) | 14 | 18 | 15.89 | 19.83 | 0.27 | 0.35 |
| Latency (s) | 0.015 | 0.015 | 0.018 | 0.016 | 0.021 | 0.18 |
| Throughput (ops/s) | 66 | 76 | 56 | 66 | 50 | 64 |

**Table 8.** Performance figures for Full Threshold (2 and 3 parties) and Shamir with 3 parties for the fixed point exp/log operations.

**Discussion:** Just as for the trigonometric functions we can compare our work to that of Cheon et al. [9] using homomorphic encryption. They perform can perform exponentiation operations in about 164 ms (not amortized), whereas we can perform an exponentiation, with a known public base, in about 2.5 ms (under our 3 parties Shamir based setting). It is worth noting that Dimitrov et al. [15] also provided implementations for the exponent function, using alternative ways to represent these rational numbers, using MPC. However, it is difficult to draw direct comparisons with this later work as they target passively secure MPC, whereas we focus on actively secure MPC.

## Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. N66001-15-C-4070, and by the FWO under an Odysseus project GOH9718N.

## References

1. Aliasgari, M., Blanton, M., Zhang, Y., Steele, A.: Secure computation on floating point numbers. In: NDSS 2013. The Internet Society (Feb 2013)
2. Aly, A., Keller, M., Orsini, E., Rotaru, D., Scholl, P., Smart, N.P., Wood, T.: SCALE and MAMBA documentation (2018), `https://homes.esat.kuleuven.be/~nsmart/SCALE/`
3. Bayatbabolghani, F., Blanton, M., Aliasgari, M., Goodrich, M.: Poster: Secure computations of trigonometric and inverse trigonometric functions
4. Bayatbabolghani, F., Blanton, M., Aliasgari, M., Goodrich, M.: Secure fingerprint alignment and matching protocols. arXiv preprint arXiv:1702.03379 (2017)
5. Beaver, D.: Foundations of secure interactive computing. In: Feigenbaum, J. (ed.) CRYPTO'91. LNCS, vol. 576, pp. 377–391. Springer, Heidelberg (Aug 1992)
6. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., López, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (Oct 2008)
7. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Garay, J.A., Prisco, R.D. (eds.) SCN 10. LNCS, vol. 6280, pp. 182–199. Springer, Heidelberg (Sep 2010)
8. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 35–50. Springer, Heidelberg (Jan 2010)
9. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full rns variant of approximate homomorphic encryption. Cryptology ePrint Archive, Report 2018/931 (2018), `https://eprint.iacr.org/2018/931`
10. Cybernetica SA: Sharemind (2018), `https://sharemind.cyber.ee`
11. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 285–304. Springer, Heidelberg (Mar 2006)
12. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (Sep 2013)

13. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 247–264. Springer, Heidelberg (Aug 2003)
14. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (Aug 2012)
15. Dimitrov, V., Kerik, L., Krips, T., Randmets, J., Willemson, J.: Alternative implementations of secure real numbers. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 16. pp. 553–564. ACM Press (Oct 2016)
16. Hart, J.F.: Computer Approximations. Krieger Publishing Co., Inc., Melbourne, FL, USA (1978)
17. Kamm, L.: Privacy-preserving statistical analysis using secure multi-party computation. Ph.D. thesis, University of Tartu, Estonia (2015)
18. Kamm, L., Willemson, J.: Secure floating-point arithmetic and private satellite collision analysis. Cryptology ePrint Archive, Report 2013/850 (2013), http://eprint.iacr.org/2013/850
19. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 158–189. Springer, Heidelberg (Apr / May 2018)
20. Keller, M., Rotaru, D., Smart, N.P., Wood, T.: Reducing communication channels in MPC. In: Catalano, D., De Prisco, R. (eds.) SCN 18. LNCS, vol. 11035, pp. 181–199. Springer, Heidelberg (Sep 2018)
21. Kerik, L., Laud, P., Randmets, J.: Optimizing MPC for robust and scalable integer and floating-point arithmetic. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D.S., Brenner, M., Rohloff, K. (eds.) FC 2016 Workshops. LNCS, vol. 9604, pp. 271–287. Springer, Heidelberg (Feb 2016)
22. Liedel, M.: Secure distributed computation of the square root and applications. In: Ryan, M.D., Smyth, B., Wang, G. (eds.) Information Security Practice and Experience. pp. 277–288. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
23. Maurer, U.: Secure multi-party computation made simple. Discrete Applied Mathematics 154(2), 370–381 (2006)
24. Medina, H.A.: A sequence of polynomials for approximating arctangent. The American Mathematical Monthly 113(2), 156–161 (2006), http://www.jstor.org/stable/27641866
25. Ning, C., Xu, Q.: Multiparty computation for modulo reduction without bit-decomposition and a generalization to bit-decomposition. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 483–500. Springer, Heidelberg (Dec 2010)
26. Ning, C., Xu, Q.: Constant-rounds, linear multi-party computation for exponentiation and modulo reduction with perfect security. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 572–589. Springer, Heidelberg (Dec 2011)
27. Zhang, Y., Steele, A., Blanton, M.: PICCO: a general-purpose compiler for private distributed computation. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 13. pp. 813–826. ACM Press (Nov 2013)

## A Polynomial and Padé Approximations

Tables in this appendix provide the concrete polynomials for the Polynomial/Padé approximations used by our protocols. The tables were extracted from Hart's

Computer Approximations [16]. We follow the same nomenclature as the book's author. That is to say, scientific notation where each of the polynomial coefficients, is given by its Significand ($s$) and its coefficient Magnitude ($m$). Let $c_i$, be the coefficient of term $i$, for all $i \in P(x)$. Then $c_i$ can be obtained by calculating $m_i \cdot 10^{s_i}$. This is true for all polynomials in this Appendix.

$\boldsymbol{P_{3307}(X)}$**:** The polynomial that is used to approximate the function $\texttt{sin}(x)$ on the interval $[0, \frac{\pi}{2}]$. The absolute error of the approximation is given by:

$$\left|\texttt{sin}(x) - x \cdot P_{3307}(x^2)\right| < 10^{-20.19} \text{ for } x \in [0, \frac{\pi}{2}].$$

| | s | **Magnitudes (m)** of $\mathbf{P_{3307}}$ |
|---|---|---|
| 0 | 1 | +0.15707 96326 79489 66192 31314 989 |
| 1 | 0 | -0.64596 40975 06246 25365 51665 255 |
| 2 | -1 | +0.79692 62624 61670 45105 15876 375 |
| 3 | -2 | -0.46817 54135 31868 79164 48035 89 |
| 4 | -3 | +0.16044 11847 87358 59304 30385 5 |
| 5 | -5 | -0.35988 43235 20707 78156 5727 |
| 6 | -7 | +0.56921 72920 65732 73962 4 |
| 7 | -9 | -0.66880 34884 92042 33722 |
| 8 | -11 | +0.60669 10560 85201 792 |
| 9 | -13 | -0.43752 95071 18174 8 |
| 10 | -15 | +0.25002 85418 9303 |

$\boldsymbol{P_{3508}(X)}$**:** The polynomial that is used to approximate the function $\texttt{cos}(x)$ on the interval $[0, \frac{\pi}{2}]$. The absolute error, for this polynomial is given by:

$$\left|\texttt{cos}(x) - P_{3508}(x^2)\right| < 10^{-23.06} \text{ for } x \in [0, \frac{\pi}{2}].$$

| | s | **Magnitudes (m)** of $\mathbf{P_{3508}}$ |
|---|---|---|
| 0 | 0 | +0.99999 99999 99999 99999 99914 771 |
| 1 | 0 | -0.49999 99999 99999 99999 91637 437 |
| 2 | -1 | +0.41666 66666 66666 66653 10411 988 |
| 3 | -2 | -0.13888 88888 88888 88031 01864 15 |
| 4 | -4 | +0.24801 58730 15870 23300 45157 |
| 5 | -6 | -0.27557 31922 39332 25642 1489 |
| 6 | -8 | +0.20876 75698 16541 25915 59 |
| 7 | -10 | -0.11470 74512 67755 43239 4 |
| 8 | -13 | +0.47794 54394 06649 917 |
| 9 | -15 | -0.15612 26342 88277 81 |
| 10 | -18 | +0.39912 65450 7924 |

***$P_{5102}(X)$* and *$Q_{5102}(X)$*:** These are the polynomials we use to calculate the Padé approximation for the function $\mathtt{arctan}(x)$ on the $[0, \mathtt{tan}(\pi/4)]$ interval. Note that, $\mathtt{tan}(\pi/4) = 1$. We can express the relative error for this approximation as:

$$\left|\frac{\mathtt{arctan}(x) - x \cdot \frac{P_{5102(x^2)}}{Q_{5102}(x^2)}}{\mathtt{arctan}(x)}\right| < 10^{-22.69}.\text{ for } x \in [0, \tan\pi/4].$$

| | s | **Magnitudes (m)** of $\mathbf{P_{5102}}$ | s | **Magnitudes (m)** of $\mathbf{Q_{5102}}$ |
|---|---|---|---|---|
| 0 | 5 | +0.21514 05962 60244 19331 93254 468 | 5 | +0.21514 05962 60244 19331 93298 234 |
| 1 | 5 | +0.73597 43380 28844 42408 14980 706 | 5 | +0.80768 78701 15592 48851 76713 209 |
| 2 | 6 | +0.10027 25618 30630 27849 70511 863 | 6 | +0.12289 26789 09278 47762 98743 322 |
| 3 | 5 | +0.69439 29750 03225 23370 59765 503 | 5 | +0.97323 20349 05355 56802 60434 387 |
| 4 | 5 | +0.25858 09739 71909 90257 16567 793 | 5 | +0.42868 57652 04640 80931 84006 664 |
| 5 | 4 | +0.50386 39185 50126 65579 37791 19 | 5 | +0.10401 13491 56689 00570 05103 878 |
| 6 | 3 | +0.46015 88804 63535 14711 61727 227 | 4 | +0.12897 50569 11611 09714 11459 55 |
| 7 | 2 | +0.15087 67735 87003 09877 17455 528 | 2 | +0.68519 37831 01896 80131 14024 294 |
| 8 | -1 | +0.75230 52818 75762 84445 10729 539 | 1 | +0.1 |

***$P_{2524}(X)$* and *$Q_{2523(X)}$*:** These are the polynomials that are used to calculate the Padé approximation for the function $\mathtt{log}_2(x)$ on the interval $[0.5, 1]$. The relative error for this approximation is given by:

$$\left|\frac{\mathtt{log}_2(x) - \frac{P_{2524(x)}}{Q_{2524}(x)}}{\mathtt{log}_2(x)}\right| < 10^{-8.32} \text{ for } x \in [0.5, 1].$$

| | s | **Magnitudes (m)** of $\mathbf{P_{2524}}$ | s | **Magnitudes (m)** of $\mathbf{Q_{2524}}$ |
|---|---|---|---|---|
| 0 | 1 | -0.20546 66719 51 | 0 | +0.35355 34252 77 |
| 1 | 1 | -0.88626 59939 1 | 1 | +0.45451 70876 29 |
| 2 | 1 | +0.61058 51990 15 | 1 | +0.64278 42090 29 |
| 3 | 1 | +0.48114 74609 89 | 1 | +0.1 |

***$P_{1045}(X)$*:** We use this polynomial, to calculate the Padé approximation for the function $\mathtt{exp}(2, x)$ on the interval $[0, 1]$. The relative error of the approximation is given by:

$$\left|\frac{\mathtt{exp}(2, x) - P_{1045}(x)}{\mathtt{exp}(2, x)}\right| < 10^{-12.11} \text{ for } x \in [0, 1].$$

| | s | **Magnitudes (m)** of $\mathbf{P_{1045}}$ |
|---|---|---|
| 0 | 1 | +0.10000 00077 44302 1686 |
| 1 | 0 | +0.69314 71804 26163 82779 5756 |
| 2 | 0 | +0.24022 65107 10170 64605 384 |
| 3 | -1 | +0.55504 06862 04663 79157 744 |
| 4 | -2 | +0.96183 41225 88046 23749 77 |
| 5 | -2 | +0.13327 30359 28143 78193 29 |
| 6 | -3 | +0.15510 74605 90052 57397 8 |
| 7 | -4 | +0.14197 84739 97656 06711 |
| 8 | -5 | +0.18633 47724 13796 7076 |