Secure Floating-Point Arithmetic and Private Satellite Collision Analysis

Liina Kamm[1,2] and Jan Willemson[1]

[1] Cybernetica AS**, Mäealuse 2/1, 12618 Tallinn, Estonia
{liina,janwil}@cyber.ee
[2] University of Tartu, Institute of Computer Science, Liivi 2, 50409 Tartu, Estonia

Abstract. In this paper we show that it is possible and, indeed, feasible to use secure multiparty computation for calculating the probability of a collision between two satellites. For this purpose, we first describe basic floating-point arithmetic operators (addition and multiplication) for multiparty computations. The operators are implemented on the Sharemind secure multiparty computation engine. We discuss the implementation details, provide methods for evaluating example elementary functions (inverse, square root, exponentiation of e, error function). Using these primitives, we implement a satellite conjunction analysis algorithm and give benchmark results for the primitives as well as the conjunction analysis itself.

1 Introduction

The Earth is orbited by nearly 7,000 spacecraft[3], orbital debris larger than 10 centimeters are routinely tracked and their number exceeds 21,000 [4]. It is understandable that countries do not want to reveal orbital information about their more strategic satellites. On the other hand, satellites are a big investment and it is in every satellite owner's interest to keep their property intact. Currently, approximate information about the whereabouts and orbits of satellites is available. These data can be analysed to predict collisions and hopefully react to the more critical results.

However, in 2009, two communications satellites belonging to the US and Russia collided in orbit [22]. Based on the publicly available orbital information, the satellites were not supposed to come closer than half a kilometer to each other at the time of the collision. As there are many orbital objects and satellite

** This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. Distribution Statement A (Approved for Public Release, Distribution Unlimited)

[3] NSSDC Master Catalog, NASA. http://nssdc.gsfc.nasa.gov/nmc/, last accessed December 16, 2013

[4] Orbital Debris Program Office, NASA. http://orbitaldebris.jsc.nasa.gov/faqs.html, last accessed December 16, 2013

operators, an all-to-all collaboration between parties is infeasible. However, public data is precise enough to perform a pre-filter and exclude all pairs of objects that cannot collide at least within a given period of time (e.g., 5 days). Once the satellite pairs with a sufficiently high collision risk have been found, the satellite operators should exchange more detailed information and determine if a collision is imminent and decide if the trajectory of either object should be modified.

We show that secure multiparty computation (SMC) can be used as a possible solution for this problem. We propose a secure method that several satellite operators can jointly use for determining if the orbital objects operated by them will collide with each other. Secure multiparty computation allows satellite operators to secret share data about their satellites so that no party can see the individual values, but collision analysis can still be conducted. The solution provides cryptographic guarantees for the confidentiality of the input trajectories.

To be able to implement the collision analysis algorithm in a privacy preserving way, we first provide floating point arithmetic for general multiparty computations and present an implementation of floating point operations in an SMC setting based on secret sharing. We build the basic arithmetic primitives (addition and multiplication), develop example elementary functions (inverse, square root, exponentiation of e, error function), and present benchmarks for the implementations. As a concrete instantiation of the underlying SMC engine, we have chosen Sharemind [7], as it provides both excellent performance and convenient development tools [17]. However, all the algorithms developed are applicable for general SMC platforms in which the necessary technical routines are implemented. Using these available resources, we finally implemented one possible collision probability computation algorithm and we give the benchmark results for this implementation.

2 Preliminaries

2.1 Secure Multiparty Computation

In this subsection, we give an overview of secure multiparty computation.

Secure multiparty computation (SMC) with n parties $P_1, \ldots, P_n$ is defined as the computation of a function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ so that the result is correct and the input values of all the parties are kept private. Every party P_i will provide an input value x_i and learn only the result value y_i. For some functions f and some input values it might be possible to deduce other parties' inputs from the result, but in the case of data aggregation algorithms, it is generally not possible to learn the inputs of other parties from the result.

In this paper we concentrate on SMC methods based on secret sharing—also called *share computing* techniques. Share computing uses *secret sharing* for the storage of data.

Definition 1. *Let s be the secret value. An algorithm $\mathbf{S}$ defines a k-*out-of-*n* threshold secret sharing scheme*, if it computes $\mathbf{S}(s) = (s_1, \ldots, s_n)$ and the following conditions hold [27]:*

1. ***Correctness:*** *s is uniquely determined by any k shares from $\{s_1, \ldots, s_n\}$ and there exists an algorithm $\mathbf{S}'$ that efficiently computes s from these k shares.*
2. ***Privacy:*** *having access to any $k-1$ shares from $\{s_1, \ldots, s_n\}$ gives no information about the value of s, i.e., the probability distribution of $k-1$ shares is independent of s.*

We use $[\![s]\!]$ to denote the shared value, i.e., the tuple $(s_1, \ldots, s_n)$.

The data storage process with three SMC parties is illustrated in Figure 1. Data are collected from *data donors* and sent to the three SMC parties called *miners*. A data donor distributes the data into n shares using secret sharing and sends one share of each value to a single miner. This separation of nodes into input nodes (donors) and SMC nodes (miners) is useful, as it does not force every party in the information system to run SMC protocols. This reduces the complexity of participating in the computation and makes the technology more accessible.

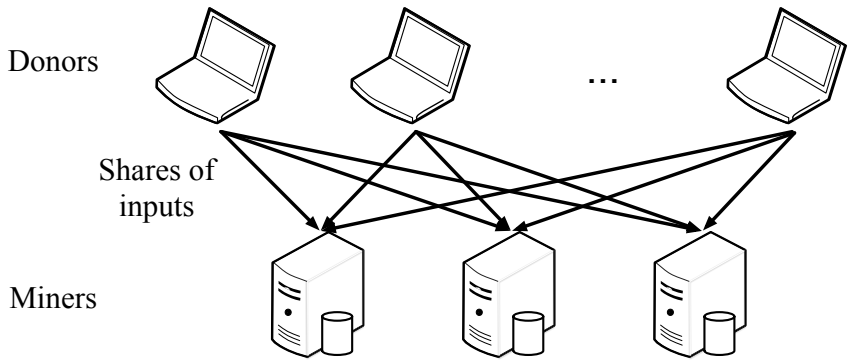

Fig. 1. The input nodes connect to all SMC nodes and store their data using secret sharing.

After the data has been stored, the SMC nodes can perform computations on the shared data. Notice that none of the SMC nodes can reconstruct the input values thanks to the properties of secret sharing. We need to preserve this security guarantee during computations. This is achieved by using secure multiparty computation protocols that specify which messages the SMC nodes should exchange in order to compute new shares of a value that corresponds to the result of an operation with the input data.

Figure 2 shows the overall structure of secure multiparty computations. Assume, that the SMC nodes have shares of input values u and v and want to compute $w = u \odot v$ for some operation $\odot$. They run the SMC protocol for operation $\odot$ which gives each SMC node one share of the result value w. Note

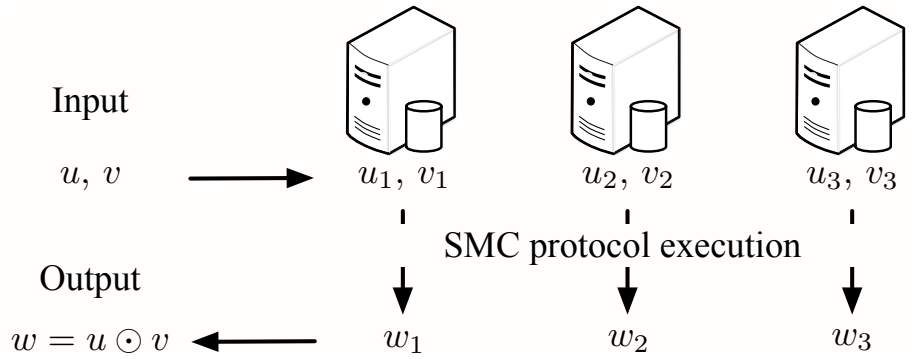

Fig. 2. The SMC nodes can run SMC protocols to compute useful results from secret-shared data.

that the protocols do not leak information about the input values u and v. For details and examples on how this can be achieved, see the classical works on SMC [29,5,13,9].

After computations have been finished, the results are published to the client of the computation. The SMC nodes send the shares of the result values to the client node that reconstructs the real result from the shares, see Figure 3. Note that it is important not to publish the results when they could still leak information about the inputs. The algorithms running in the SMC nodes must be audited to ensure that they do not publish private inputs. However, note that even if some nodes publish their result shares too early, they cannot leak data unless their number exceeds the threshold k.

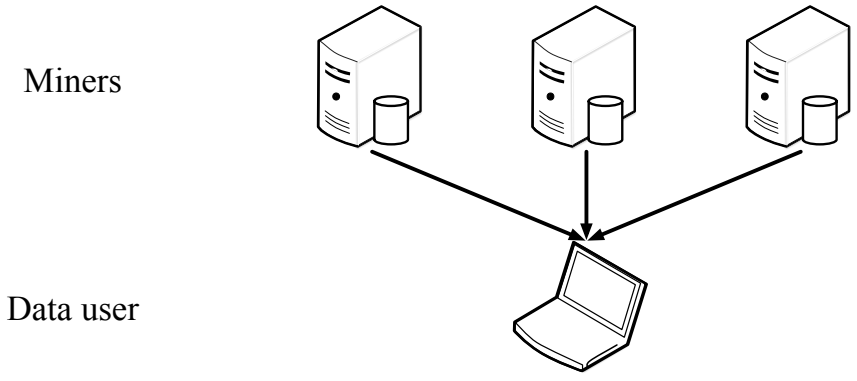

Fig. 3. Results are published after the computation is complete.

Several secure multiparty computation frameworks have implementations [21,15,28,14]. While the majority of implementations are research-oriented, it is expected that more practical systems will be created as the technology matures.

2.2 The Sharemind Platform

SHAREMIND [7] is an implementation of privacy-preserving computation technology based on secure multiparty computation. In the most recent version of SHAREMIND, the users can choose which underlying secure multiparty computation method suits them best. For each available method, some basic operations (e.g addition, multiplication) have been implemented. More complex operations (e.g division) often rely on these basic operations. If the basic operations are universally composable, it is possible to use them to implement the complex operations without additional effort. Knowing this, we can design protocols for these complex operations without having a specific secure multiparty computation method in mind. Hence, the operations can be used with any kind of underlying method as long as it supports the basic operations that we need for the protocol. Some complex protocols, on the other hand, require us to know what the underlying schemas are and must, therefore, be redesigned for each underlying SMC method.

The default protocol suite in SHAREMIND relies on the three party additive secret sharing scheme working on n-bit integers, $n \in \{8, 16, 32, 64\}$, i.e., each value x is split into three shares x_1, x_2, x_3 so that $x_1 + x_2 + x_3 = x \bmod 2^n$ and $[\![x]\!] = (x_1, x_2, x_3)$. In this article, this is the underlying scheme we use for testing the protocols and for benchmarking results. This is also the scheme for which we have implemented targeted optimisations that are dependent on the format in which the data are stored.

The default protocol suite of SHAREMIND has formal security proofs in the passive security model allowing one corrupt miner. This ensures that the security of inputs is guaranteed as long as the miner servers do not deviate from the protocols and do not disclose the contents of their private database. The protocols are also proven to be universally composable, meaning that we can run them sequentially or in parallel without losing the privacy guarantees. For more details on the security model of SHAREMIND, please refer to the PhD thesis [6]. Note that since then, the primitive protocols of SHAREMIND have been updated to be more efficient [8].

3 Collaborative satellite collision probability analysis

3.1 Motivation and state of the art

The first confirmed and well-studied collision between a satellite and another orbital object was the collision of the French reconnaissance satellite Cerise with a fragment of an Ariane rocket body in 1996 [24].

The first accidental collision between two intact satellites happened on February 10th, 2009 when the US communications satellite Iridium 33 collided with a decommissioned Russian communications satellite Kosmos 2251 [22]. The two orbital planes intersected at a nearly right angle, resulting in a collision velocity of more than 11 km/s. Not only did the US lose their working satellite, the collision left two distinct debris clouds floating in Earth's orbit. Even though the number of tracked debris is already high, an even larger amount is expected in case of a body-to-body hit. Preliminary assessments indicate that the orbital lifetimes of many of the debris are likely to be tens of years, and as the debris gradually form a shell about the Earth, further collisions with new satellites may happen.

This event was the fourth known accidental collision between two catalogued objects and produced more than 1000 pieces of debris [18]. The previous collisions were between a spacecraft and a smaller piece of debris and did not produce more than four pieces of new debris each.

The US Space Surveillance Network (part of the US Strategic Command) is maintaining a catalogue of orbital objects, like satellites and space debris, since the launch of early satellites like the Sputnik. Some of the information is available on the Space-Track.org[5] web page for registered users. However, publicly available orbital data is often imprecise. For example, an analysis based on the publicly available orbital data determined that the Iridium 33 and Kosmos 2251 would come no closer to each other than half a kilometer at the time of the collision. This was not among the top predicted close approaches for that report nor was it the top predicted close approach for any of the Iridium satellites for the coming week [18].

This, however, was not a fault in the analysis software but rather the quality of the data. As a result of how the orbital data are generated, they are of low fidelity, and are, therefore, of limited value during analysis. If more precise data were acquired directly from satellite operators and governments, and the collision analysis process were agreed upon, these kinds of collisions could be avoided [19].

3.2 Underlying math

Algorithm 1 describes how to compute the probability of collision between two spherical objects and is based on [2,3,26]. As input, the satellite operators give the velocity vectors $\boldsymbol{v} \in \mathbb{R}^3$, covariance matrices $\mathbf{C} \in \mathbb{R}^{3\times3}$, position vectors $\boldsymbol{p} \in \mathbb{R}^3$, and radii $R \in \mathbb{R}$ of two space objects. First, on line 2, we define an orthogonal coordinate system in terms of the velocity vectors and their difference. The resulting $\boldsymbol{j}$-$\boldsymbol{k}$ plane is called the conjunction plane. We consider space objects as spheres but as all the uncertainty in the problem is restricted to the conjunction plane, we can project this sphere onto the conjunction plane (line 4) and treat the problem as 2-dimensional [2].

Next, we diagonalise the matrix C to choose a basis for the conjunction plane where the basis vectors are aligned with the principal axes of the ellipse (lines 5-

[5] Space-Track.org, `https://www.space-track.org`, last accessed December 16, 2013

ALGORITHM 1: Collision analysis using public data

Input: Velocity vectors $\boldsymbol{v}_a, \boldsymbol{v}_b \in \mathbb{R}^3$, covariance matrices $\mathbf{C}_a, \mathbf{C}_b \in \mathbb{R}^{3\times 3}$, position vectors $\boldsymbol{p}_a, \boldsymbol{p}_b \in \mathbb{R}^3$, radii $R_a, R_b \in \mathbb{R}$

Output: Probability p of a collision between the two described objects a and b

1 Set $\boldsymbol{v}_r \leftarrow \boldsymbol{v}_b - \boldsymbol{v}_a$

2 Set $\boldsymbol{i} \leftarrow \frac{\boldsymbol{v}_r}{|\boldsymbol{v}_r|}$, $\boldsymbol{j} \leftarrow \frac{\boldsymbol{v}_b \times \boldsymbol{v}_a}{|\boldsymbol{v}_b \times \boldsymbol{v}_a|}$ and $\boldsymbol{k} \leftarrow \boldsymbol{i} \times \boldsymbol{j}$

3 Set $\mathbf{Q} \leftarrow [\boldsymbol{j} \;\; \boldsymbol{k}]$

4 Set $\mathbf{C} \leftarrow \mathbf{Q}^T(\mathbf{C}_a + \mathbf{C}_b)\mathbf{Q}$

5 Set $(\boldsymbol{u}, \boldsymbol{v}) \leftarrow$ Eigenvectors($\mathbf{C}$)

6 Set $(\sigma_x^2, \sigma_y^2) \leftarrow$ Eigenvalues($\mathbf{C}$)

7 Set $\sigma_x \leftarrow \sqrt{\sigma_x^2}$ and $\sigma_y \leftarrow \sqrt{\sigma_y^2}$

8 Set $\boldsymbol{u} \leftarrow \frac{\boldsymbol{u}}{|\boldsymbol{u}|}$ and $\boldsymbol{v} \leftarrow \frac{\boldsymbol{v}}{|\boldsymbol{v}|}$

9 Set $\mathbf{U} \leftarrow [\boldsymbol{u} \;\; \boldsymbol{v}]$

10 Set $\begin{bmatrix} x_m \\ y_m \end{bmatrix} \leftarrow \mathbf{U}^T\mathbf{Q}^T(\boldsymbol{p}_b - \boldsymbol{p}_a)$

11 Set $R \leftarrow R_a + R_b$

12

$$\text{Set } p \leftarrow \frac{1}{2\pi\sigma_x\sigma_y}\int_{-R}^{R}\int_{-\sqrt{R^2-x^2}}^{\sqrt{R^2-x^2}} \exp\left[\frac{-1}{2}\left[\left(\frac{x-x_m}{\sigma_x}\right)^2 + \left(\frac{y-y_m}{\sigma_y}\right)^2\right]\right] dydx$$

13 **return** p

9). We go on to project the centre of the hardbody ($\boldsymbol{p_b} - \boldsymbol{p_a}$) onto the encounter plane and then put it in the eigenvector basis (line 10) to gain the coordinates (x_m, y_m). Finally, the probability of collision is computed on line 12 [3].

Vector and matrix operations To implement the analysis given in Algorithm 1, we need to compute the unit vector, dot product and the cross product for vectors of length 3. Furthermore, we need matrix multiplication for two-dimensional matrices and, also, the computation of the eigensystem for 3×3 matrices. These operations require multiplication, addition, division and the computation of square root.

Computing integrals In addition to common vector and matrix operations, Algorithm 1 also contains the two-dimensional probability equation for the combined spherical object (line 12)

$$p \leftarrow \frac{1}{2\pi\sigma_x\sigma_y}\int_{-R}^{R}\int_{-\sqrt{R^2-x^2}}^{\sqrt{R^2-x^2}} \exp\left[\frac{-1}{2}\left[\left(\frac{x-x_m}{\sigma_x}\right)^2 + \left(\frac{y-y_m}{\sigma_y}\right)^2\right]\right] dydx \;\;,$$

where R is the combined radius of the bodies, (x_m, y_m) is the centre of the hardbody in the encounter plane, and σ_x and σ_y are the lengths of the semi-principle axes. Similarly to the article [3], we use Simpson's one-third rule for

numerical integration of this double integral. As a result, we get the following equation:

$$p \approx \frac{\Delta x}{3\sqrt{8\pi}\sigma_x}\left[f(0) + f(R) + \sum_{i=1}^{n} 4f((2i-1)\Delta x) + \sum_{i=1}^{n-1} 2f(2i\Delta x)\right] \ ,$$

where $\Delta x = \frac{R}{2n}$ and the integrand is

$$f(x) = \left[\operatorname{erf}\left(\frac{y_m + \sqrt{R^2 - x^2}}{\sqrt{2}\sigma_y}\right) + \operatorname{erf}\left(\frac{-y_m + \sqrt{R^2 - x^2}}{\sqrt{2}\sigma_y}\right)\right] \times \left[\exp\left(\frac{-(x + x_m)^2}{2\sigma_x^2}\right) + \exp\left(\frac{-(-x + x_m)^2}{2\sigma_x^2}\right)\right] \ .$$

For further details, see [3]. To compute the collision probability, we require multiplication, addition, division, square root, exponentiation of e and the computation of the error function.

3.3 Privacy-preserving solution using secure multiparty computation

The locations and orbits of communications satellites are sensitive information and governments or private companies might not be willing to reveal these data. One possible solution is to use a trusted third party who will gather all the data and perform the analysis. This, however, still requires the satellite owners to reveal their information to an outside party. While this could be done within one country, it is doubtful that different governments are willing to disclose the orbits of their satellites to a single third party that is inevitably connected with at least one country.

We propose using secure multiparty computation instead of a trusted third party. Figure 4 describes the general idea of our solution. The satellite operators choose three independent parties as the data hosts. These organisations will either host the secure computation system themselves or outsource it to a secure hosting provider. The hosts must be chosen such that the probability of collusion between them is negligible. Next, the operators secret share their data and upload the shares to the three hosts. Collision analysis is performed in collaboration between the tree hosting parties and the satellite operators can query the results of the analysis.

4 Secure floating point operations

4.1 Related work

Integer data domains are not well suited for scientific and engineering computations, where rational and real numbers allow achieving a much greater precision.

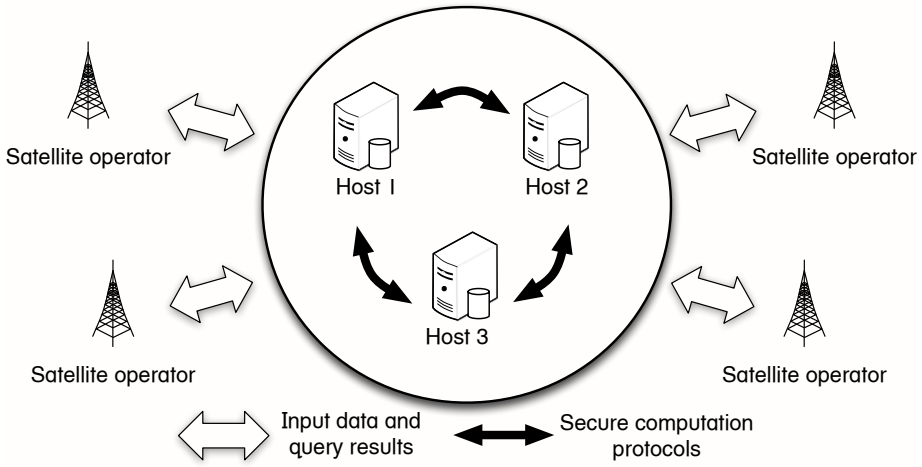

Fig. 4. Privacy-preserving collision analysis using secure multiparty computation.

Catrina *et al.* took the first steps in adapting real number arithmetic to secure multi-party computation by developing secure fixed point arithmetic and applying it to linear programming [12,10,11]. In 2011, Franz and Katzenbeisser proposed a solution for the implementation of floating point arithmetic for secure signal processing [16]. However, their approach relies on two-party computations working over garbled circuits, and they do not provide the actual implementation nor any benchmarking results. In 2012, 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 [4].

4.2 Representation of Floating Point Numbers

The most widely used standard for floating point arithmetic is IEEE 754 [1]. To reach our goal of implementing the arithmetic, we will also make use of Donald Knuth's more generic presentation given in Section 4.2 of the book [20], and ideas used in FPGA implementations [23].

The majority of the available approaches split the representation of a floating point number N into several parts.

- *Radix* (*base*) b (usually 2 or 10).
- *Sign* s (e.g., having the value 0 or 1 corresponding to signs plus and minus, respectively).
- *Significand* f representing the significant digits of N. This number is assumed to belong to a fixed interval like $[1, 2)$ or $[1/b, 1)$.
- *Exponent* e showing the number of places the radix point of the significand needs to be shifted in order to produce the number N.

– *Bias* (*excess*) q is a fixed number used to make the representation of e non-negative; hence one would actually use E such that $e = E - q$ for storing the exponent. For example, for IEEE 754 double precision arithmetic it is defined that $q = 1023$, $e \in [-1022, 1023]$ and $E \in [1, 2046]$. The value $E = 0$ is reserved for representing $N = 0$ and some other very small values. The value $E = 2047$ is reserved for special quantities like infinity and Not a Number (NaN) occurring as a result of illegal operations (e.g. $0/0$ or $\sqrt{-1}$).

Using these notations, we have a generic representation for floating point numbers

$$N = (-1)^s \cdot f \cdot b^{E-q} .$$

The IEEE 754 standard is clearly targeted towards explicitly available data, where the computing entity has access to all the bits of the representation. This allows for several optimisations, e.g., putting the sign, significand and exponent together into one machine word, or leaving the leading digit of the significand out of the representation, as it can be assumed to be 1 and, hence, carries no information. Before real computations on such numbers can start, a special procedure called *unpacking* is required, accompanied by its counterpart *packing* to restore the representation after the computation is over.

In case of a secret-shared representation, however, access to the bits is non-trivial and involves a lot of computation for bit extraction [7,8]. An alternative would be storing the values shared bitwise, but this would render the underlying processor arithmetic unusable and we would need to reimplement all the basic arithmetic operations on these shares. At this point we do not rule this possibility out completely, but this approach needs good justification before actual implementation.

In order to save time on bit extraction in the course of packing and unpacking, we will store the crucial parts of the floating point numbers (sign, significand and exponent) in separately shared values s, e and f that have 8, 16 and 32 bits, respectively. We also allow double precision floating point values, where the sign and exponent are the same as for 32-bit floating point values, but the significand is stored using 64 bits.

The (unsigned) significand f will represent the real number $\overline{f} = f/2^{32}$. In other words, the highest bit of f corresponds to $1/2$, the second one to $1/4$, etc. In order to obtain real floating point behaviour, we require that the representation is normalised, i.e., that the highest bit of f is always 1 (unless the number to be represented is zero, in which case $f = 0$, s corresponds to the sign of the value, and the biased exponent $e = 0$). Hence, for all non-zero floats we can assume that $f \in [1/2, 1)$. With our single precision floating point values, we achieve 32-bit significand precision, which is somewhere in between IEEE single and double precision. The double precision floating point values achieve 64-bit significand precision, which is more than the IEEE double precision.

The sign s can have two values: 1 and 0 denoting plus and minus, respectively. Note that we are using a convention opposite to the standard one to implement specific optimisations. As we are using a 16-bit shared variable e to store the

ALGORITHM 2: Protocol for multiplying two n-bit ($n \in \{32, 64\}$) shared floating point numbers $[\![N'']\!] \leftarrow \mathsf{MultFloat}([\![N]\!], [\![N']\!])$

Input: Shared floating point values $[\![N]\!]$ and $[\![N']\!]$
Output: Shared floating point value $[\![N'']\!]$ such that $N'' = N \cdot N'$

1 Set $[\![s'']\!] \leftarrow \mathsf{Eq}([\![s]\!], [\![s']\!])$
2 Set $[\![e'']\!] \leftarrow [\![e + e']\!]$
3 Cast the shared significands $[\![f]\!]$ and $[\![f']\!]$ to $2 \cdot n$ bits and multiply to obtain $[\![f'']\!]_{2n}$
4 Truncate $[\![f'']\!]_{2n}$ down to n bits
5 Compute the highest bit λ of f'' as $[\![\lambda]\!] \leftarrow [\![f'']\!] \gg (n-1)$ // *Normalize the significand*
6 **if** $[\![\lambda]\!] = 0$ **then**
7 $\quad [\![f'']\!] \leftarrow [\![f'']\!] \ll 1$
8 $\quad [\![e'']\!] \leftarrow [\![e'']\!] - 1$
9 **end**
10 **return** $[\![N'']\!] = ([\![s'']\!], [\![f'']\!], [\![e'']\!])$

exponent, we do not need to limit the exponent to 11 bits like in the standard representation. Instead we will use 15 bits (we do not use all 16 because we want to keep the highest bit free) and we introduce a bias of $2^{14} - 1$.

In a typical implementation of floating point arithmetic, care is taken to deal with exponent under- and overflows. In general, all kinds of exceptional cases (including under- and overflows, division by zero, $\sqrt{-1}$) are very problematic to handle with secret-shared computations, as issuing any kind of error message as a part of control flow leaks information (e.g., the division by zero error reveals the divisor).

We argue, that in many practical cases it is possible to analyse the format of the data, based on knowledge about their contents. This allows the analysts to know that the data fall within certain limits and construct their algorithms accordingly. This makes the control flow of protocols independent of the inputs and preserves privacy in all possible cases. For example, consider the scenario, where the data is gathered as 32-bit integers and then the values are cast to floating point numbers. Thus the exponent cannot exceed 32, which means that we can multiply 2^9 of such values without overflowing the maximal possible exponent 2^{14}. This approach to exceptions is basically the same as Strategy 3 used in the article [16].

4.3 Multiplication

The simplest protocol for floating point arithmetic is multiplication, as the underlying representation of floating point numbers is multiplicative. The routine for multiplying two secret-shared floating point numbers is presented in Algorithm 2. In the following, we will use the notation $[\![N]\!]$ to denote the triple $([\![s]\!], [\![f]\!], [\![e]\!])$. Throughout this and several following protocols we will make use of primitives described in [8]. Recall, that we do not check for any exponent under- or overflows during multiplication.

The general idea of the algorithm is natural. The sign of the product is determined as an equality bit of the arguments (line 1) and can be computed as

$$\mathsf{Eq}([\![s]\!],[\![s']\!]) = [\![(1-s)\cdot(1-s')]\!] + [\![s\cdot s']\!] = \\ 1 - [\![s]\!] - [\![s']\!] + 2\cdot[\![s\cdot s']\!] \ .$$

The exponent of the result is the sum of the exponents of the arguments (line 2) and the significands are multiplied. However, if we simply multiplied the representations of the significands, we would obtain a wrong result. For example, our reference framework SHAREMIND uses n-bit integers ($n \in \{32, 64\}$) and arithmetic modulo 2^n. Hence, out of the $2n$ product bits, we obtain the lowest n after implicit modular reduction. These, however, are the least significant bits for out floating point significand representation, and we actually need the highest n bits instead. Therefore, we need to cast our additively shared n-bit values temporarily to $2n$ bits and truncate the product to n bits again. Both of these operations are non-trivial, as we cannot ignore the overflow bits while casting up nor can we just throw away the last bits when truncating, and these cases are rather troublesome to handle obliviously.

For the casting on line 3, keep in mind that as SHAREMIND uses modular arithmetic, then while casting the significand up, we need to account for the overflow error generated by the increase of the modulus. For instance, if we are dealing with 32-bit floats, the significand is 32 bits. When we cast it to 64 bits, we might get an overflow of up to 2 bits. To deal with this problem, we check obliviously whether our resulting 64 bit float is larger than 2^{32} and 2^{33} and adjust the larger value accordingly.

Truncation on line 4 can be performed by deleting the lower n bits of each share. This way we will lose the possible overflow that comes from adding the lower bits. This will result in a rounding error, so we check whether the lower n bits actually generate overflow. We do this similarly to the up-casting process—by checking whether the sum of the three shares of the lower n bits is larger than 2^{32} and 2^{33}. Then we truncate the product of the significands and obliviously add 0, 1 or 2 depending on the overflow we received as the result of the comparison.

For non-zero input values we have $f, f' \in [1/2, 1)$, and hence $f'' = f \cdot f' \in [1/4, 1)$ (note that precise truncation also prevents us from falling out of this interval). If the product is in the interval $[1/4, 1/2)$, we need to normalise it. We detect the need for normalisation by the highest bit of f'' which is 0 exactly if $f \in [1/4, 1/2)$. If so, we need to shift the significand one position left (line 7) and reduce the exponent by 1 (line 8).

The SHAREMIND system does not hide the execution flow of protocols, so whenever *if-then* statements have secret-shared values in the condition, an oblivious choice is used in the implementation. For example, the statement on line 6 is implemented using the following oblivious choice:

$$[\![f'']\!] \leftarrow [\![\lambda \cdot f'']\!] + [\![(1-\lambda)\cdot(f'' \ll 1)]\!] \ , \\ [\![e'']\!] \leftarrow [\![e'']\!] - 1 + [\![\lambda]\!] \ .$$

ALGORITHM 3: Protocol for obliviously choosing a between two shared values $[\![x]\!]$ and $[\![y]\!]$ based on a shared condition $[\![c]\!]$

Input: Shared values $[\![x]\!]$, $[\![y]\!]$ and shared condition $[\![c]\!]$
Output: Shared value $[\![x]\!]$ if $[\![c]\!]$ is true (1) or $[\![y]\!]$ if $[\![c]\!]$ is false (0)
1 Set $[\![z]\!] \leftarrow [\![c \cdot x]\!] + [\![(1-c) \cdot y]\!]$
2 **return** z

A general way for obliviously choosing between two inputs is given in Algorithm 3. In case of floating point numbers, we perform three oblivious choices separately for each component s, f and e. This is faster and more precise than multiplying floating point numbers with one and zero.

4.4 Addition

Because of the multiplicative representation of floating point numbers, addition is more difficult to implement than multiplication and requires us to consider more special cases.

The overall structure of the addition routine is presented in Algorithm 4. In order to add two numbers, their decimal points need to be aligned. The size of the required shift is determined by the difference of the exponents of the inputs. We perform an oblivious switch of the inputs based on the bit $[\![e']\!] \overset{?}{\geq} [\![e]\!]$, so that in the following we can assume that $[\![e']\!] \geq [\![e]\!]$ (line 1). Our reference architecture SHAREMIND uses n-bit integers to store the significands, hence if $[\![e']\!] - [\![e]\!] \geq n$, adding N is efficiently as good as adding 0 and we can just output $[\![N']\!]$ (line 2). Unfortunately, in the implementation, we cannot halt the computation here, as the control flow must be data-independent, but this check is needed for the following routines to work correctly. Instead of returning, the result of the private comparison is stored and an oblivious choice is performed at the end of the protocol to make sure that the correct value is returned.

On line 5, we shift one of the significands to align the decimal points and adjust the exponent. Moving on to the *if-else* statement on lines 6–22, the operation performed on the significands depends on the sign bits. If they are equal (lines 6–13), we need to add the significands, and otherwise subtract them. When adding the significands, the result may exceed 1, so we may need to shift it back by one position. In order to achieve this functionality, we use temporary casting to $2n$ bits similarly to the process in Algorithm 2 for multiplication.

If the sign bits differ, we need to subtract the smaller significand from the larger one. As a result, we may end up with a value less than $1/2$, in which case we need to normalise the value (lines 15–22). The *if-else* statement on lines 17–20 can be computed as

$$[\![f'']\!] = [\![b \cdot (f' - f)]\!] + [\![(1-b) \cdot (f - f')]\!] = 2 \cdot [\![b \cdot (f' - f)]\!] + [\![f]\!] - [\![f']\!] \ .$$

Note that the right shift used on line 5 of Algorithm 4 must be done obliviously as the value has to be shifted by a private number of bits. Oblivious shift

ALGORITHM 4: Protocol for adding two n-bit ($n \in \{32, 64\}$) shared floating point numbers $[\![N'']\!] \leftarrow \mathsf{AddFloat}([\![N]\!], [\![N']\!])$

Input: Shared floating point values $[\![N]\!]$ and $[\![N']\!]$
Output: Shared floating point value $[\![N'']\!]$ such that $N'' = N + N'$

1 Assume $[\![e' - e]\!] \geq 0$ (if not, switch $[\![N]\!]$ and $[\![N']\!]$)
2 **if** $[\![e' - e]\!] \geq n$ **then** // *N' is so much larger that effectively $N = 0$*
3 **return** $[\![N']\!]$
4 **end**
5 Set $[\![f]\!] \leftarrow [\![f \gg (e' - e)]\!]$ and $[\![e'']\!] \leftarrow [\![e']\!]$
6 **if** $s = s'$ **then**
7 $[\![s'']\!] \leftarrow [\![s]\!]$
8 Cast $[\![f]\!]$ and $[\![f']\!]$ to $2n$ bits
9 $[\![f'']\!] \leftarrow [\![f]\!] + [\![f']\!]$
10 $[\![b]\!] = [\![f'']\!] \gg n$ // *Let the $(n+1)$st bit of $[\![f'']\!]$ be $[\![b]\!]$*
11 $[\![f'']\!] \leftarrow [\![f'' \cdot (1 - b)]\!] + [\![f'']\!]$
12 $[\![f'']\!] \gg 1$, $[\![e'']\!] \leftarrow [\![e'']\!] + [\![b]\!]$ // *Correcting the overflow*
13 Cast $[\![f'']\!]$ down to n bits
14 **else**
15 Let $[\![b]\!] \leftarrow [\![f']\!] \overset{?}{\geq} [\![f]\!]$
16 $[\![s'']\!] \leftarrow \mathsf{Eq}([\![s']\!], [\![b]\!])$
17 **if** $[\![b]\!]$ **then**
18 $[\![f'']\!] \leftarrow [\![f']\!] - [\![f]\!]$ // *This can give a non-normalized significand!*
19 **else**
20 $[\![f'']\!] \leftarrow [\![f]\!] - [\![f']\!]$ // *This can give a non-normalized significand!*
21 **end**
22 Normalize $[\![N'']\!] = ([\![s'']\!], [\![f'']\!], [\![e'']\!])$
23 **end**
24 **return** $[\![N'']\!]$

right can be accomplished by using Algorithm 5. First, we compute all the possible n right-shifted values in parallel. Next, we extract the bits of the shift using the $\mathsf{BitExtr}$ routine from [8]. Recall that $p \in [0, n-1]$, so we obtain $\log_2(n)$ bits. Next, based on these $\log_2(n)$ bits we compose a characteristic vector consisting of n shared bits, of which most have value 0 and exactly one has value 1. The value 1 occurs in the p-th position of the vector. Finally, the final result can be obtained by computing the dot product between the vector of shifted values and the characteristic vector.

Normalisation of the significand and the exponent on line 22 of Algorithm 4 can be performed by using Algorithm 6. It works with a similar principle as Algorithm 5 and performs an oblivious selection from a vector of elements containing all the possible output values. The selection vector is computed using the MSNZB (most significant non-zero bit) routine from [8] which outputs n additively shared bits, at most one being equal to 1 in the position corresponding to the highest non-zero bit of the input. The vectors from which we obliviously

ALGORITHM 5: Protocol for shifting a shared value $[\![f]\!]$ of n bits right by a shared number of positions $[\![p]\!]$

Input: Shared values $[\![f]\!]$ and $[\![p]\!]$, where $0 \leq p \leq n-1$
Output: Shared value $[\![f \gg p]\!]$

1 Compute in parallel $[\![f_i]\!] = [\![f]\!] \gg i$, $i = (0, \ldots, n-1)$
2 $\overline{[\![p]\!]} \leftarrow \mathsf{BitExtr}[\![p]\!]$
3 Compute the characteristic bits $[\![p_i]\!]$ $(i = 0, \ldots, n-1)$ based on $\overline{[\![p]\!]}$
4 **return** $\sum_{i=0}^{n-1} [\![f_i \cdot p_i]\!]$

ALGORITHM 6: Protocol for normalizing an n-bit shared floating point value $[\![N]\!]$

Input: Shared floating point value $[\![N]\!] = ([\![s]\!], [\![f]\!], [\![e]\!])$, where the highest non-zero bit of f is on p-th position from the highest bit
Output: Shared floating point value $[\![N']\!] = ([\![s]\!], [\![f']\!], [\![e']\!])$ such that $f' = f \ll p$ and $e' = e - p$. If $N = 0$ then $N' = 0$

1 $\overline{[\![g]\!]} \leftarrow \mathsf{MSNZB}([\![f]\!])$
2 Compute in parallel $[\![f_i]\!] = [\![f]\!] \ll i$, $i = (0, \ldots, n-1)$
3 $[\![f']\!] \leftarrow \sum_{i=0}^{n-1} [\![f_i \cdot \boldsymbol{g}_{n-1-i}]\!]$
4 $[\![e']\!] \leftarrow \sum_{i=0}^{n-1} [\![(e-i) \cdot \boldsymbol{g}_{n-1-i}]\!]$
5 **return** $[\![N']\!] = ([\![s]\!], [\![f']\!], [\![e']\!])$

select consist of all the possible shifts of the significands (lines 2 and 3) and the corresponding adjustments of the exponents (line 4).

5 Elementary Functions

In this section, we present algorithms for four elementary functions, namely inversion, square root, exponentiation of e and error function. All of them are built using Taylor series expansion and we also give a version using the Chebyshev polynomial for inversion and the exponentiation of e. Polynomial evaluations are sufficiently robust for our purposes. They allow us to compute the functions using only additions and multiplications, and do so in a data-independent manner. Taylor series expansion also gives us a convenient way to have several trade-offs between speed and precision.

5.1 Inversion

We implement division by first taking the inverse of the divisor and then multiplying the result with the dividend. We compute the inverse by using Taylor series:

$$\frac{1}{x} = \sum_{n=0}^{\infty} (1-x)^n \quad .$$

ALGORITHM 7: Protocol for taking the inverse of an n-bit shared floating point value $[\![N]\!]$ with precision p using Taylor series

Input: Shared floating point value $[\![N]\!] = ([\![s]\!], [\![f]\!], [\![e]\!])$
Output: Shared floating point value $[\![N']\!]$ such that $[\![N']\!] = \frac{1}{[\![N]\!]}$

1 Let $[\![S]\!] = ([\![s_S]\!], [\![f_S]\!], [\![e_S]\!])$ and $[\![S']\!] = ([\![s_{S'}]\!], [\![f_{S'}]\!], [\![e_{S'}]\!])$ be shared floats
2 Set $[\![s_S]\!] \leftarrow 1$, $[\![f_S]\!] \leftarrow [\![f]\!]$ and $[\![e_S]\!] \leftarrow 0$
3 Set $[\![S]\!] \leftarrow 1 - [\![S]\!]$
4 Evaluate the Taylor series $[\![S']\!] \leftarrow \sum_{n=0}^{p} [\![S]\!]^n$
5 Set $[\![e']\!] \leftarrow [\![e_{S'}]\!] - [\![e]\!]$
6 **return** $[\![N']\!] = ([\![s]\!], [\![f_S]\!], [\![e']\!])$

Algorithm 7 describes how to take the inverse of a shared n-bit float using Taylor series. As the inverse of a value N converges best in the interval $(0, 1]$, we separate the significand $f \in [\frac{1}{2}, 1)$ from the given floating point number and use the following equation:

$$\frac{1}{f \cdot 2^e} = \frac{1}{f} \cdot 2^{-e} \quad . \tag{1}$$

To achieve this, we compose a new floating point value $[\![S]\!]$ based on the significand $[\![f]\!]$ on line 2. Note that we make this new float always positive for optimisation purposes and, at the end of the protocol, we reintroduce the sign s from the input float $[\![N]\!]$ (line 6). When we have evaluated the Taylor series for inverse, we correct the exponent based on equality (1) by subtracting the previous exponent from the exponent of the sum $[\![S']\!]$ (line 5).

As SHAREMIND works fastest with parallelised computations, we do as much of the Taylor series evaluation in parallel as possible. This parallelisation is described in Algorithm 8. Variations of this algorithm can be adopted for the specific polynomial evaluations where needed, but this algorithm gives a general idea of how the process works. We use Algorithm 8 to parallelise polynomial evaluation in all elementary functions we implement.

We need to compute powers of $[\![x]\!]$ from $[\![x^1]\!]$ to $[\![x^p]\!]$. To do this in parallel, we fill vector $[\![\boldsymbol{u}]\!]$ with all the powers we have already computed $[\![x^1]\!], \ldots, [\![x^h]\!]$ and the second vector $[\![\boldsymbol{v}]\!]$ with the highest power of $[\![x]\!]$ we have computed $[\![x^h]\!]$. When we multiply these vectors element-wise, we get powers of $[\![x]\!]$ from $[\![x^{h+1}]\!]$ to $[\![x^{2h}]\!]$ (lines 6 to 16). If the given precision p is not a power of 2, however, we also need to do one extra parallel multiplication to get the rest of the powers from $[\![x^h]\!]$ to $[\![x^p]\!]$. This is done on lines 18 to 29. If the precision is less than 1.5 times larger than the highest power, it is not necessary to add elements to the vector $[\![\boldsymbol{u}]\!]$ (checked on line 20).

The *if-else* statements in Algorithm 8 are based on public values, so there is no need to use oblivious choice to hide which branch is taken. The summation of vector elements on line 35 is done in parallel as much as possible.

ALGORITHM 8: Algorithm for parallel polynomial evaluation

Input: Shared floating point value $[\![x]\!]$, precision p, coefficients for the polynomial $a_0, \ldots, a_p$

Output: Shared floating point value $[\![x']\!]$ such that $[\![x']\!] = \sum_{n=0}^{p} a_n \cdot x^n$

Let $[\![\boldsymbol{y}]\!] = ([\![y_0]\!], \ldots, [\![y_p]\!])$ // *For storing the powers of* x
Set $[\![y_0]\!] \leftarrow 1$, $[\![y_1]\!] \leftarrow [\![x]\!]$ and $[\![y_2]\!] \leftarrow [\![x^2]\!]$
Let $[\![\boldsymbol{u}]\!]$, $[\![\boldsymbol{v}]\!]$ and $[\![\boldsymbol{w}]\!]$ be shared vectors of size 1
Set $[\![u_0]\!] \leftarrow [\![x]\!]$, and $[\![w_0]\!] \leftarrow [\![x^2]\!]$
Let $h = 2$ denote the highest power of x we have computed thus far
while $2h \leq p$ **do**
    Resize $[\![\boldsymbol{u}]\!]$, $[\![\boldsymbol{v}]\!]$, $[\![\boldsymbol{w}]\!]$ to size h
    for $i = h/2 + 1$ **to** h **do**
        Set $[\![u_{i-1}]\!] \leftarrow [\![x^i]\!]$
    end
    Fill $[\![\boldsymbol{v}]\!]$ with $[\![x^h]\!]$
    Multiply in parallel $[\![w_i]\!] \leftarrow [\![u_i \cdot v_i]\!]$, $i = (0, \ldots, h-1)$ // *To get* $[\![x^{h+1}]\!], \ldots, [\![x^{2h}]\!]$
    for $i = h + 1$ **to** $2h$ **do**
        Set $[\![y_i]\!] \leftarrow [\![x^i]\!]$
    end
    Set $h \leftarrow 2h$
end
if $2h > p$ **then**
    Resize $[\![\boldsymbol{u}]\!]$ and $[\![\boldsymbol{v}]\!]$ to size $p - h$
    if $(p > h + h/2)$ **then**
        for $i = h/2 + 1$ **to** p **do**
            Set $[\![u_{i-1}]\!] \leftarrow [\![x]\!]^i$
        end
    end
    Fill $[\![\boldsymbol{v}]\!]$ with $[\![x^h]\!]$
    Resize $[\![\boldsymbol{w}]\!]$ to size $p - h$
    Multiply in parallel $[\![w_i]\!] \leftarrow [\![u_i \cdot v_i]\!]$, $i = (0, \ldots, p-h-1)$ // *To get* $[\![x^{h+1}]\!], \ldots, [\![x^p]\!]$
    for $i = h + 1$ **to** p **do**
        Set $[\![y_i]\!] \leftarrow [\![x^i]\!]$
    end
end
for $i = 0$ **to** p **do**
    Set $[\![y_i]\!] \leftarrow [\![y_i \cdot a_i]\!]$ // *Multiplication with the given constants*
end
Set $[\![x']\!]$ as the sum of the elements of $[\![\boldsymbol{y}]\!]$
return $[\![x']\!]$

As the Taylor series evaluation requires at least 32 summation terms to achieve the precision we need, we also offer an alternative algorithm for computing inverse using the corresponding approximated Chebyshev polynomial [25].

ALGORITHM 9: Protocol for taking the inverse of an n-bit shared floating point value $[\![N]\!]$ using the Chebyshev polynomial

Input: Shared floating point value $[\![N]\!] = ([\![s]\!], [\![f]\!], [\![e]\!])$
Output: Shared floating point value $[\![N']\!]$ such that $[\![N']\!] = \frac{1}{[\![N]\!]}$

1 Let $[\![S]\!]$ and $[\![S']\!]$ be shared floats $([\![s_S]\!], [\![f_S]\!], [\![e_S]\!])$ and $([\![s_{S'}]\!], [\![f_{S'}]\!], [\![e_{S'}]\!])$, respectively
2 Set $[\![s_S]\!] \leftarrow 1$, $[\![f_S]\!] \leftarrow [\![f]\!]$ and $[\![e_S]\!] \leftarrow 0$
3 Set $[\![S']\!] \leftarrow -12.9803173971914[\![S^7]\!] + 77.6035205859554[\![S^6]\!] - 201.355944395853[\![S^5]\!] + 296.100609596203[\![S^4]\!] - 269.870620651938[\![S^3]\!] + 156.083648732173[\![S^2]\!] - 55.9375000000093[\![S]\!] + 11.3566017177974$
4 Set $[\![e']\!] \leftarrow [\![e_{S'}]\!] - [\![e]\!]$
5 **return** $[\![N']\!] = ([\![s]\!], [\![f_S]\!], [\![e']\!])$

Algorithm 9 is similar to Algoritm 7, but instead of evaluating Taylor series, we compute the Chebyshev polynomial using the parallelisation from Algorithm 8.

5.2 Square Root

Square root is the most complex of the elementary functions we implemented. The Taylor series is given by the equation:

$$\sqrt{x} = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}(2n-2)!}{(1-2n-2)((n-1)!)^2 4^{n-1}} \cdot (x-1)^{n-1} \ .$$

The convergence interval of the series for square root is $(0, 2)$. Thus, we will compute the square root of the significand and multiply it with a correcting exponential term:

$$\sqrt{f \cdot 2^e} = \sqrt{f} \cdot 2^{e/2} \ . \tag{2}$$

Algorithm 10 shows how square root is computed based on equation (2). We start by creating a float containing the absolute value of the significand $[\![f]\!]$ of the original value N and go on by evaluating the Taylor series for $[\![f]\!] - 1$. Line 5 marks the beginning of the computation of the exponent. We first divide the exponent by 2 by shifting the exponent right once and adding it to the exponent $[\![e_S]\!]$ we received as a result of the Taylor series evaluation. On line 7 we find the lowest bit of the exponent to determine whether it is even or odd. Now, we still have to compensate for the bit we lost during division (lines 8–14). If the exponent is even, we do nothing, otherwise, we multiply the intermediate result $[\![N']\!]$ by $\sqrt{2}$ or $\sqrt{\frac{1}{2}}$ depending on the sign of the exponent. The *if-then* statements on lines 8–14 and 9–13 need to be performed obliviously, as both $[\![b]\!]$ and $[\![e]\!]$ are secret shared values.

ALGORITHM 10: Protocol for finding the square root of a floating point number $[\![N']\!] \leftarrow \sqrt{[\![|N|]\!]}$ with precision p

Input: Shared floating point value $[\![N]\!]$
Output: Shared floating point value $[\![N']\!]$ such that $N' = \sqrt{|N|}$ with precision p

Let $[\![f]\!]$ be the significand of $[\![|N|]\!]$
Set $[\![x]\!] \leftarrow [\![f]\!] - 1$
Let $[\![S]\!]$ be a shared float $([\![s_S]\!], [\![f_S]\!], [\![e_S]\!])$
Evaluate the Taylor series $[\![S]\!] \leftarrow \sum_{n=1}^{p} \frac{(-1)^{n-1}(2n-2)!}{(1-2n-2)((n-1)!)^2 4^{n-1}} \cdot [\![x^{n-1}]\!]$
Set $[\![e']\!] \leftarrow [\![e_S]\!] + ([\![e]\!] \gg 1)$ // *Divide* $[\![e]\!]$ *by* 2
Set $[\![N']\!] \leftarrow (1, [\![f_S]\!], [\![e']\!])$
Let $[\![b]\!]$ be the lowest bit of $[\![e]\!]$ // *For determining whether* $[\![e]\!]$ *is odd or even*
if $[\![b]\!] = 1$ **then**
    if $[\![e]\!] \geq 0$ **then**
        Set $[\![N']\!] \leftarrow [\![N']\!] \cdot \sqrt{2}$
    else
        Set $[\![N']\!] \leftarrow [\![N']\!] \cdot \sqrt{\frac{1}{2}}$
    end
end
return $[\![N']\!]$

ALGORITHM 11: Protocol for raising e to the power of float value $[\![N]\!]$ with precision p using Taylor series

Input: Shared floating point value $[\![N]\!] = ([\![s]\!], [\![f]\!], [\![e]\!])$
Output: Shared floating point value $[\![N']\!]$ such that $[\![N']\!] = e^{[\![N]\!]}$ with precision p

Evaluate the Taylor series $[\![N']\!] \leftarrow \sum_{n=0}^{p} \frac{1}{n!} \cdot [\![N^n]\!]$
return $[\![N']\!] = ([\![s]\!], [\![f_S]\!], [\![e']\!])$

5.3 Exponentiation of e

Algorithm 11 describes the exponentiation of e using the Taylor series:

$$e^x = \sum_{n=0}^{\infty} \frac{1}{n!} \cdot x^n \ .$$

Unfortunately, the larger $[\![N]\!]$ is, the more we have to increase precision p to keep the result of the Taylor series evaluation from deviating from the real result too much. This, however, increases the working time of the algorithm significantly. To solve this problem, we implemented another algorithm using the separation of the integer and fractional parts of the power $[\![N]\!]$ and using the Chebyshev polynomial to approximate the result [25]. This solution is described in Algorithm 12.

In this case, we exponentiate e using the knowledge that $e^x = (2^{\log_2 e})^x = 2^{(\log_2 e) \cdot x}$. Knowing this, we can first compute $y = (\log_2 e) \cdot x \approx 1.44269504 \cdot x$

ALGORITHM 12: Protocol for exponentiating e to the power of an n-bit shared floating point value $[\![N]\!]$ using the separation of integer and fractional parts of $[\![N]\!]$

Input: Shared floating point value $[\![N]\!] = ([\![s]\!], [\![f]\!], [\![e]\!])$
Output: Shared floating point value $[\![N']\!] = e^{[\![N]\!]}$

1 Set $[\![y]\!] \leftarrow 1.44269504 \cdot [\![N]\!]$ // *Change bases*
2 Denote the integer part of $[\![y]\!]$ by $[\![[y]]\!]$ and the fractional part by $[\![\{y\}]\!]$
3 Set $[\![[y]]\!] \leftarrow [\![(-1)^{1-s} \cdot ((f \gg (n-e)) + (1-s))]\!]$ // *Find the integer part*
4 Set $[\![\{y\}]\!] \leftarrow [\![y]\!] - [\![[y]]\!]$ // *Find the fractional part (always positive)*
5 Let $2^{[\![[y]]\!]} = ([\![s']\!], [\![f']\!], [\![e']\!])$
6 Set $[\![s']\!] \leftarrow 1$, $[\![f']\!] \leftarrow 100\ldots0$, and $[\![e']\!] \leftarrow [\![[y]]\!] + 1$
7 Set $2^{[\![\{y\}]\!]} \leftarrow 0.0136839828938349[\![\{y\}^4]\!] + 0.0517177354601992[\![\{y\}^3]\!] + 0.241621322662927[\![\{y\}^2]\!] + 0.692969550931914[\![\{y\}]\!] + 1.00000359714456$
8 **return** $[\![N']\!] = 2^{[\![[y]]\!]} \cdot 2^{[\![\{y\}]\!]}$

(line 1) and then 2^y. Notice, that $2^y = 2^{[y]+\{y\}} = 2^{[y]} \cdot 2^{\{y\}}$, where $[y]$ denotes the integer part and $\{y\}$ the fractional part of y.

First we need to separate the integer part from the shared value $[\![y]\!] = ([\![s]\!], [\![f]\!], [\![e]\!])$. Let n be the number of bits of f. In this case, $[\![[|y|]]\!] = [\![f \gg (n-e)]\!]$ (line 3) is true if and only if $[\![e]\!] \leq n$. On the other hand, if $[\![e]\!] > n$, then $[\![y]\!] \geq 2^{[\![e]\!]-1} > 2^{n-1}$ would also hold and we would be trying to compute a value that is not smaller than $2^{2^{n-1}}$. Hence, we consider the requirement $[\![e]\!] \leq n$ to be reasonable.

Note, however, that if we want to use the Chebyshev polynomial to compute $2^{\{y\}}$, we need to use two different polynomials for the positive and negative case. We would like to avoid this if possible for optimisation, hence, we want to keep $\{y\}$ always positive. This is easily done if y is positive, but causes problems if y is negative. So if y is negative and we want to compute the integer part of y, we compute it in the normal way and add 1 (line 3) making its absolute value larger than that of y. Now, when we subtract the gained integer to gain the fraction on line 4, we always get a positive value. This allows us to only evaluate one Chebyshev polynomial, thus giving us a slight performance improvement.

We construct the floating point value $2^{[\![[y]]\!]}$ by setting the corresponding significand to be $100\ldots0$ and the exponent to be $[\![[y]]\!] + 1$. As $[\![\{y\}]\!] \in [0, 1]$ holds for the fractional part of y and we have made sure that the fraction is always positive, we can compute $2^{[\![\{y\}]\!]}$ using the Chebyshev polynomial on line 7. In this domain, these polynomials will give a result that is accurate to the fifth decimal place. Finally, we multiply $2^{[\![[y]]\!]} \cdot 2^{[\![\{y\}]\!]}$ to gain the result.

5.4 Error function

We implemented the error function by using the following Taylor series:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n}{n!(2n+1)} \cdot x^{2n+1} \quad .$$

ALGORITHM 13: Protocol for computing the error function for shared floating point value $[\![N]\!]$ with precision p

Input: Shared floating point value $[\![N]\!] = ([\![s]\!], [\![f]\!], [\![e]\!])$
Output: Shared floating point value $[\![N']\!]$ such that $[\![N']\!] = \mathrm{erf}([\![N]\!])$ with precision p

1 **if** $[\![e]\!] \geq 3$ **then**
2 Set $N' \leftarrow (-1)^{1-[\![s]\!]}$
3 **else**
4 Evaluate the Taylor series $[\![N']\!] \leftarrow \frac{2}{\sqrt{\pi}} \sum_{n=0}^{p} \frac{(-1)^n}{n!(2n+1)} \cdot [\![N^{2n+1}]\!]$
5 **end**
6 **return** $[\![N']\!]$

Algorithm 13 describes how to compute the error function using Taylor series.

Note, that the Taylor series for the error function works on odd powers of x. To use the parallelisation trick from Algorithm 8, we first compute $y = x^2$ and we get the following equation:

$$\begin{aligned}\mathrm{erf}(x) &= \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n}{n!(2n+1)} \cdot x^{2n+1} = \frac{2}{\sqrt{\pi}} \cdot x \sum_{n=0}^{\infty} \frac{(-1)^n}{n!(2n+1)} \cdot x^{2n} \\ &= \frac{2}{\sqrt{\pi}} \cdot x \sum_{n=0}^{\infty} \frac{(-1)^n}{n!(2n+1)} \cdot y^n \ .\end{aligned}$$

From this, it is easy to see that we can evaluate the Taylor series for y and then multiply the result by x.

Note that, in addition, we check whether the exponent is larger or smaller than 3, which shows us whether $x \in [-2, 2]$ and based on the private comparison, we obliviously set the result as ± 1 according to the original sign $[\![s]\!]$ (line 2). We perform this approximation because at larger values, more cycles are needed for a sufficiently accurate result.

6 Performance of floating point operations

6.1 Benchmark results

We implemented the secure floating point operations described in this paper on the Sharemind 3 secure computation system. We built the implementation using the three-party protocol suite already implemented in the Sharemind system. Several low-level protocols were developed and optimised to simplify the use and conversion of shared values with different bit sizes. Secure floating point operations are programmed as compositions of low-level integer protocols.

To measure the performance of the floating point operations we deployed the developed software on three servers connected with fast network connections. More specifically, each of the servers used contains two Intel X5670 2.93 GHz CPUs with 6 cores and 48 GB of memory. The network connections are

point-to-point 1 Gb/s connections. We note that during the experiments, peak resource usage for each machine did not exceed 2 cores and 1 GB of RAM. The peak network usage varied during the experiments, but did not exceed 50 Mb/s. Therefore, we believe that the performance can be improved with further optimisations.

Integer operations on SHAREMIND perform more efficiently on multiple inputs [8]. Therefore, we also implemented floating point operations as vector operations. Unary operations take a single input vector and binary operations take two input vectors of an equal size. Both kinds of operations produce a vector result. We performed the benchmarks on vectors of various sizes to estimate how much the size of the vector affects the processing efficiency.

For each operation and input size, we performed 10 iterations of the operation and computed the average. The only exceptions to this rule were operations based on Taylor series or Chebyshev polynomials running on the largest input vector. Their running time was extensively long so we performed only a few tests for each of them. In the presentation, such results are marked with an asterisk.

Tables 1 and 2 present the benchmarking results for floating point numbers with 32-bit significands and 64-bit significands, respectively. We see that with smaller input sizes, there is no difference in the running time of single and double precision operations. However, for larger input vector sizes, the additional communication starts to slow down the computation as the network channel gets saturated.

Addition and multiplication are the primitive operations with addition being about five times slower than multiplication. With single precision floating point numbers, both can achieve the speed of a thousand operations per second if the input vectors are sufficiently large. Based on the multiplication primitive we also implemented multiplication and division by a public constant. For multiplication, there was no gain in efficiency, because we currently simply classify the public factor and use the private multiplication protocol.

However, for division by a public constant we can have serious performance gains as we just invert the divisor publicly and then perform a private multiplication. Furthermore, as our secret floating point representation is based on powers of two, we were able to make a really efficient protocol for multiplying and dividing with public powers of two by simply modifying the exponent. As this is a local operation, it requires no communication, and is, therefore, very fast.

Although we isolated the network during benchmarking, there was enough fluctuation caused by the flow control to slightly vary the speeds of similar operations. Although private multiplication, multiplication and division by constant are performed using the same protocol, their speeds vary due to effects caused by networking.

The unary operations are implemented using either Taylor series or Chebyshev polynomials. We benchmarked both cases where they were available. In all such cases, Chebyshev polynomials provide better performance, because smaller

Table 1. Performance of single precision floating point operations

		Input size in elements				
		1	**10**	**100**	**1000**	**10000**
Private x Private y	Add	0.73 ops	7.3 ops	69 ops	540 ops	610 ops
	Multiply	2.3 ops	23 ops	225 ops	1780 ops	6413 ops
	Divide (TS)	0.08 ops	0.81 ops	6.8 ops	25.5 ops	38 ops*
	Divide (CP)	0.16 ops	1.6 ops	14 ops	59 ops	79 ops*
Private x Public y	Multiply	2.3 ops	23 ops	220 ops	1736 ops	6321 ops
	Multiply by 2^k	$5.9 \cdot 10^4$ ops	$6.1 \cdot 10^5$ ops	$4.2 \cdot 10^6$ ops	$1.1 \cdot 10^7$ ops	$1.4 \cdot 10^7$ ops
	Divide	2.3 ops	23 ops	221 ops	1727 ops	6313 ops
	Divide by 2^k	$6.4 \cdot 10^4$ ops	$6.1 \cdot 10^5$ ops	$4.1 \cdot 10^6$ ops	$1.2 \cdot 10^7$ ops	$1.4 \cdot 10^7$ ops
Private x	Find e^x (TS)	0.09 ops	0.9 ops	7.3 ops	23 ops	30 ops*
	Find e^x (CP)	0.12 ops	1.2 ops	11 ops	71 ops	114 ops
	Invert (TS)	0.09 ops	0.84 ops	6.8 ops	14.7 ops	35.7 ops*
	Invert (CP)	0.17 ops	1.7 ops	15.3 ops	55.2 ops	66.4 ops
	Square root	0.09 ops	0.85 ops	7 ops	24 ops	32 ops*
	Error function	0.1 ops	0.97 ops	8.4 ops	30 ops	39 ops

Notes: All performance values are in operations per second (ops). Operations marked with TS and CP use Taylor series or Chebyshev polynomials, respectively. For the exponent function, square root, inversion and division, the number of elements in the Taylor series is 32. For the error function, 16 elements are computed.

Table 2. Performance of double precision floating point operations

		Input size in elements				
		1	**10**	**100**	**1000**	**10000**
Private x Private y	Add	0.68 ops	6.8 ops	60 ops	208 ops	244 ops
	Multiply	2.1 ops	21 ops	195 ops	1106 ops	2858 ops
	Divide (TS)	0.08 ops	0.7 ops	3 ops	5.2 ops	12.4 ops*
	Divide (CP)	0.15 ops	1.5 ops	12 ops	23 ops	52 ops*
Private x Public y	Multiply	2.1 ops	21 ops	193 ops	1263 ops	2667 ops
	Multiply by 2^k	$3.3 \cdot 10^4$ ops	$3.2 \cdot 10^5$ ops	$2.1 \cdot 10^6$ ops	$8.5 \cdot 10^6$ ops	$1.3 \cdot 10^7$ ops
	Divide	2.1 ops	21 ops	194 ops	1223 ops	2573 ops
	Divide by 2^k	$6.4 \cdot 10^4$ ops	$6.1 \cdot 10^5$ ops	$4 \cdot 10^6$ ops	$1 \cdot 10^7$ ops	$1.2 \cdot 10^7$ ops
Private x	Find e^x (TS)	0.09 ops	0.8 ops	3.1 ops	4.5 ops	6.2 ops*
	Find e^x (CP)	0.11 ops	1.1 ops	9.7 ops	42 ops	50 ops
	Invert (TS)	0.08 ops	0.75 ops	3.6 ops	4.8 ops	10.7 ops*
	Invert (CP)	0.16 ops	1.5 ops	11.1 ops	29.5 ops	47.2 ops
	Square root	0.08 ops	0.76 ops	4.6 ops	9.7 ops	10.4 ops*
	Error function	0.09 ops	0.89 ops	5.8 ops	16 ops	21 ops*

Notes: All performance values are in operations per second (ops). Operations marked with TS and CP use Taylor series or Chebyshev polynomials, respectively. For the exponent function, square root, inversion and division, the number of elements in the Taylor series is 32. For the error function, 16 elements are computed.

polynomials are evaluated to compute the function. In general, all these operations have similar performance.

As mentioned in Subsection 5.1, division between two private floating point values is implemented by inverting the second parameter and performing a private multiplication. As inversion is significantly slower than multiplication, the running time of division is mostly dictated by the running time of inversion.

6.2 Difference in the precision of Taylor series and Chebyshev polynomials

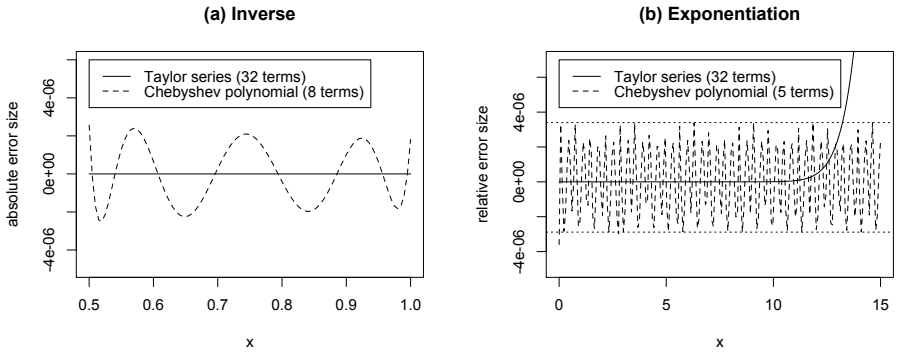

Fig. 5. Comparison of the precision of the implemented Taylor series and Chebyshev polynomials.

In Section 5, we give two algorithms for computing inverse and powers of e based on Taylor series and Chebyshev polynomials. It is easy to see from Tables 1 and 2 that using the Chebyshev polynomial versions to approximate the results gives us better performance.

Figure 5 shows the difference in the precision of these two polynomials. Panel (a) describes the absolute error that computing the inverse using both algorithms produces in the interval $[1/2, 1]$. As can be seen from the figure, the absolute error produced by the evaluation of 32 terms of the Taylor series is far less than 10^{-6}. The maximum error produced by the evaluation of 8 terms of the Chebyshev polynomial is not larger than $2.1 \cdot 10^{-6}$. The algorithm separates the exponent, then works on the significand, and, finally, subtracts the exponent. This means that the absolute error of the Chebyshev polynomial will grow as the value being inverted grows. The error of the Taylor series evaluation is so small that it will start showing itself significantly later.

Panel (b) shows similar reasoning for exponentiation with the different polynomials. As 32 elements of Taylor series for exponentiation will not be enough to produce a precise result after $x = 13$ and will produce an absolute error of

more than $2 \cdot 10^6$ at $x = 20$, we give this plot in the interval $[0, 15]$ and show the relative error instead of the absolute error. As can be seen from the figure, the relative error produced by evaluation of 5 terms of the Chebyshev polynomial always stays between $-2.9 \cdot 10^{-6}$ and $3.4 \cdot 10^{-6}$. Taylor series evaluation is more precise until $x = 13$ after which the relative error increases. This point can be pushed further by adding more terms to the Taylor series evaluation at the cost of performance.

As expected, Figure 5 indicates that using the Taylor series evaluation gives us a more precise result but, in larger algorithms, it can be a bottleneck. We propose that the choice of the algorithm be done based on the requirements of the problem being solved—if precision is important, Taylor series evaluation should be used, if speed is important, Chebyshev polynomials are the wiser choice. If the problem requires both speed and precision, it is possible to add terms to the Chebyshev polynomial to make it more precise. In addition, as the error in exponentiation using Taylor series evaluation becomes significant when x grows, Chebyshev polynomials should be used if it is expected that x will exceed 10. Another option is to raise the number of terms in Taylor series but this will, in turn, increase the running time of the algorithm.

6.3 Benefits of parallelisation

The benchmarking results confirm that our implementation of secure floating point operations is more efficient on many simultaneous inputs. The amortised cost of processing a single input or a pair of inputs is reduced as the input size grows. We performed an additional round of experiments to measure the extent of this behaviour. We ran the addition and multiplication operations with a wider range of steps to determine the optimal number of parallel additions and multiplications. We fitted the results with two linear regressions and plotted the result. The resulting plot for single-precision floating point operations is in Figure 6. The corresponding plot for double-precision numbers is in Figure 7.

Note that the running time of the secure computation protocols grows very little until a certain point, when it starts growing nearly linearly in the size of the inputs. At this point, the secure computation protocols saturate the network link and are working at their maximum efficiency. Addition is a more complex operation with more communication and therefore we can perform less additions before the network link becomes saturated. Addition takes even more communication with double-precision numbers so it is predictable that the saturation point occurs earlier than for single-precision values. Multiplication uses significantly less network communication, so its behaviour is roughly similar with both single- and double-precision numbers.

Based on these observations, we decided to use vector operations in our collision probability computation for both primitives and the main algorithm. To validate this direction, we measured the performance of the matrix product function with several sequential executions versus one or more parallel executions. In the sequential case, a number of matrix multiplications were performed one after the other. In the parallel case, they were performed simultaneously, using

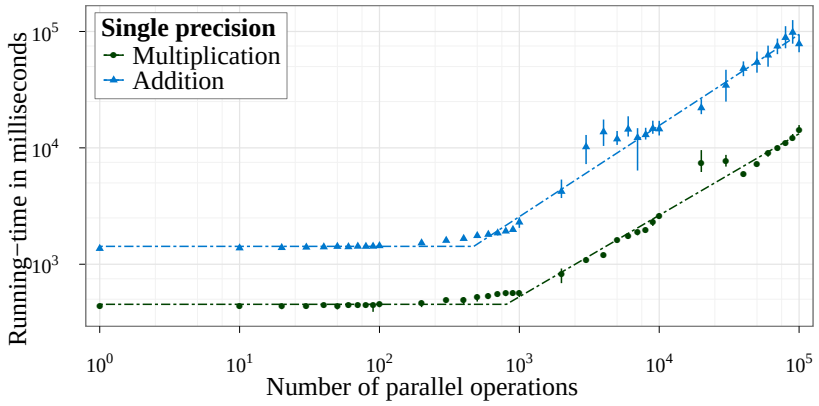

Fig. 6. The efficiency of single-precision addition and multiplication in relation to input size.

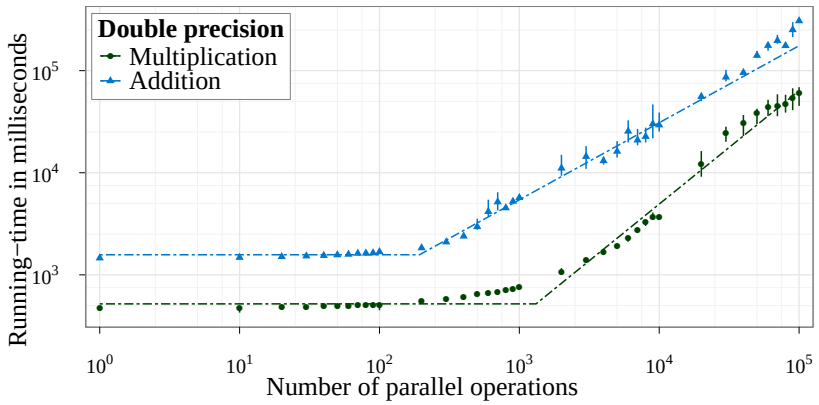

Fig. 7. The efficiency of double-precision addition and multiplication in relation to input size.

the efficiency of vector operations as much as possible. The results are given in Table 3.

Table 3. Sequential and parallel performance of the matrix product function

		Number of matrix multiplications			
		1	**10**	**100**	**1000**
3×3 matrix	Sequential	3.2 sec	31.9 sec	318.5 sec	3212.3 sec
	Parallel	3.2 sec	3.3 sec	5.2 sec	45.7 sec
5×5 matrix	Sequential	4.6 sec	46.3 sec	466 sec	4641.4 sec
	Parallel	4.6 sec	5.1 sec	20.4 sec	91.9 sec
10×10 matrix	Sequential	6.4 sec	65.9 sec	674.6 sec	6624.2 sec
	Parallel	6.6 sec	15.7 sec	152 sec	1189.2 sec

Notes: Matrix multiplication was performed using single-precision operations. For each matrix size, 10 iterations were performed and the average of the results was computed. Durations are given in seconds.

We see that, for a single input, there is no significant difference between the running time of the sequential and parallel version. This is logical, as there is nothing to vectorise. However, as the number of multiplications grows, parallel execution becomes dramatically more efficient. However, when the network saturates, the performance of the parallel case also become more linear in the number of inputs, as illustrated by the 10×10 case. Even then, the parallel case is much faster than the sequential case. This justifies our decision to use vectorisation to the highest possible degree. We note that this increase in efficiency comes at the cost of needing more memory. However, we can control the size of our inputs and balance vectorisation and memory use.

7 Implementation of collision analysis

7.1 Reusable components

We implemented the collision analysis described in Algorithm 1 for Sharemind in its language SecreC. This allows us to easily do parallel operations on vectors as it has syntax for element-wise operations on vectors. We first implemented the algorithm and its component methods for one satellite pair using as much parallelisation as possible and then went on to parallelise the solution further for n satellite pairs.

One of the most important values of our code is its reusability. The vector and matrix operations we have implemented can later be used in other algorithms similarly to the modular design of software solutions that is the norm in the industry. This makes it convenient to implement other algorithms that make use

of vector and matrix operations. In addition, the operations have already been parallelised so future developers will not have to put time into this step of the coding process in SecreC.

7.2 Vector and matrix operations

We implemented and parallelised a number of vector and matrix operations:

- Vector length, unit vector and dot product for vectors of any size;
- Cross product for vectors of size 3;
- Matrix product for two-dimensional matrices; and
- Eigensystem computation for 3×3 matrices.

We implemented simple and parallel methods for computing the unit vector and dot product for vectors of any length. We also implemented the cross product for vectors of length 3 as we are working in three-dimensional space.

Similarly, we implemented simple and parallel methods for matrix multiplication for two-dimensional matrices. We also provide a slight optimisation for the case of diagonal matrices as the covariance matrices we work with are diagonal. This does not provide a very large speedup in the case of one satellite pair but if many pairs are analysed in parallel, then the benefit is worth the implementation of such a conditional method. We also implemented the eigensystem computation for 3×3 matrices.

```
template <domain D : additive3pp>
D float64 dotProduct (D float64[[1]] x, D float64[[1]] y){
    D float64[[1]] product = x * y;
    return sum (product);
}
```

Fig. 8. Simple code for dot product

Dot product Let us take a closer look at the code of the dot product. We use this example to show the difference between the normal and the parallelised versions of the algorithm. Figure 8 shows the code for a simple dot product of two vectors $[\![\boldsymbol{x}]\!]$ and $[\![\boldsymbol{y}]\!]$. Firstly, the vectors $[\![\boldsymbol{x}]\!]$ and $[\![\boldsymbol{y}]\!]$ are multiplied element-wise and then the elements of the resulting vector $[\![\boldsymbol{product}]\!]$ are summed together.

The method from Figure 9 gets as input two matrices $[\![\mathbf{x}]\!]$ and $[\![\mathbf{y}]\!]$. We assume that these matrices are vectors that contain the data of several vectors and we assume that they have the same dimensions. The function **shape** returns a vector the length of which is determined by the dimension of the array it receives as input. In our case the variable $\boldsymbol{xShape}$ is a vector of length 2. Let the dimensions of $[\![\mathbf{x}]\!]$ and $[\![\mathbf{y}]\!]$ be m and n, then $xShape = (m, n)$, where n is the number of

```
template <domain D : additive3pp>
D float64[[1]] dotProduct (D float64[[2]] x, D float64[[2]] y){
    uint[[1]] xShape = shape (x);
    D float64[[2]] product = x * y;

    D float64[[1]] result (xShape[0]);
    D float64[[1]] productVector = reshape (product, size (product));
    result = sum (productVector, xShape[0]);
    return result;
}
```

Fig. 9. Parallelised code for dot product

elements in the vectors and m is the number of vectors. As the result, we expect a vector of m dot products. It is easy to see, that the dimensions of the input and output values of the parallel version are higher by one as compared to the simple version.

We start the parallel dot product computation similarly to the simple version—we multiply the matrices together element-wise. The SecreC language allows us to do this easily by using the multiplication sign on two arrays of the same dimensions. As the result, we receive the matrix $[\![\mathbf{product}]\!]$ that has the same dimensions as the input matrices. To use the function **sum** that allows us to sum together vector elements, we first reshape the matrix $[\![\mathbf{product}]\!]$ to a vector. This is another array manipulation method which SecreC provides us. Finally, we use the method that lets us sum vector elements together. However, in the parallel case, we use a different version of the method **sum** that lets us sum elements together by groups, giving us $xShape[0] = m$ sums as a result. The second parameter of the method **sum** must be a factor of the size of the vector, the elements of which are being summed.

The benefit in using the parallel version of the method is in the multiplication and summing sub methods. If we computed the dot product for each satellite pair as a cycle, we would get the products within one pair $[\![\boldsymbol{x_1} \cdot \boldsymbol{y_1}]\!], [\![\boldsymbol{x_2} \cdot \boldsymbol{y_2}]\!], [\![\boldsymbol{x_3} \cdot \boldsymbol{y_3}]\!]$ in parallel but would not be able to use the parallelisation for computing many pairs at once. As the multiplication and summation of floating point values is quite time consuming, parallelisation is essential.

7.3 Benchmark results

Finally, we measured the performance of the full privacy-preserving collision analysis implementation. We performed two kinds of experiments. First, we measured the running time of the algorithm on various input sizes to know if it is feasible in practice and if there are efficiency gains in evaluating many satellite pairs in parallel. Second, we profiled the execution of a single instance of collision analysis to study, what the bottleneck operations in its execution are.

The running time measurements are given in Table 4. We differentiated the results based on the precision of the floating point operations and the kind of approximation used for complex operations.

Table 4. Collision analysis performance

		Number of satellite pairs				
		1	**2**	**4**	**8**	**16**
Single precision	TS	258 (258) sec	350 (175) sec	488 (122) sec	792 (99) sec	1348 (84) sec
	CP	204 (204) sec	255 (128) sec	355 (89) sec	557 (70) sec	967 (60) sec
Double precision	TS	337 (337) sec	447 (223) sec	724 (181) sec	1127 (141) sec	2064 (129) sec
	CP	246 (246) sec	340 (170) sec	450 (113) sec	724 (91) sec	1272 (80) sec

Notes: Three iterations were performed for each experiment. First, we give the total running time and then the running time per satellite pair in parentheses. We computed 40 iterations in the integral approximation. TS means that only Taylor series were used for elementary functions. CP means that operations based on Chebyshev polynomials were used where available.

The running time for a single collision probability computation is 4-5 minutes and it is reduced 1-2 minutes with parallel executions. Consider a scenario, where we compute collision probabilities once a week and we want to get the results in a day. We could then compute the collision analysis on at least 250 satellite pairs in about 8 hours. Given that we can balance the computation load to multiple secure computation clusters, we can scale the system to process a much larger number of satellites. Also, we can use the less precise public trajectories as a preliminary filter and use the precise secure collision estimation only on the satellite pairs for which the collision probability is above a certain threshold.

We used the built-in profiling features of Sharemind to analyse the breakdown of the execution time. We computed the collision probability of a single satellite pair and measured the duration of each floating point operation in the execution. The results are given in Figures 10 and 11. Addition also includes the running time of subtraction, because they are implemented using the same protocol. Multiplication also includes division by a public value for the same reason.

When we compute collision probabilities using operations based on Taylor series, we see that division and square root are the most expensive operations, followed by the exponent function and addition. Using Chebyshev polynomials greatly reduces the importance of division and the exponent function, leaving square root as the most time-consuming operation in the algorithm.

8 Conclusions and further work

In this paper we showed how to perform satellite collision analysis in a secure multiparty setting. For this purpose, we first presented routines for implementing

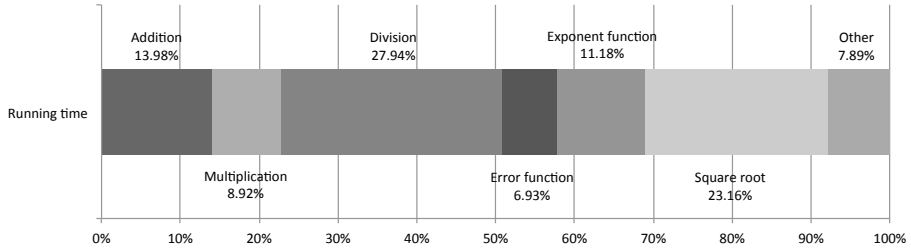

Fig. 10. Breakdown of secure floating point operation runtime in collision analysis with Taylor series.

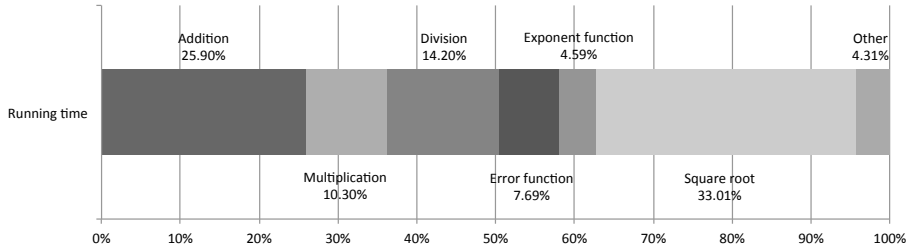

Fig. 11. Breakdown of secure floating point operation runtime in collision analysis with Chebyshev polynomials.

floating point arithmetic operations on top of a secure multiparty computation engine. The general approach is generic, but in order to obtain the efficient real implementation, platform-specific design decisions have been made. In the current paper, we based our decisions on the architecture of SHAREMIND which served as a basic platform for our benchmarks. We also showed how to implement several elementary functions and benchmarked their performance. However, by replacing the specific optimisations with generic algorithms or other specific optimisations, these floating point algorithms can be ported to other secure computation systems. We concluded by implementing the algorithm for computing the probability of satellite collision and benchmarked the performance of this implementation.

The largest impact of this work is the conclusion that it is possible and feasible to perform satellite collision analysis in a privacy preserving manner.

Acknowledgements

The authors would like to thank the RAND Corporation for research in the area of collision probability computation and Galois Inc for the use of their prototype that works on public data. The authors also thank Nigel Smart for suggesting the use of Chebyshev polynomials, Dan Bogdanov for his help with

benchmarking and Sven Laur for coming up with a neat trick for slightly speeding up exponentiation.

References

1. 754-2008 IEEE Standard for Floating-Point Arithmetic, 2008.
2. Maruthi R. Akella and Kyle T. Alfriend. Probability of collision between space objects. *Journal of Guidance, Control and Dynamics*, 23(5):769–772, 2000.
3. Salvatore Alfano. A numerical implementation of spherical object collision probability. *Journal of the Astronautical Sciences*, 53(1):103, 2005.
4. Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. Cryptology ePrint Archive, Report 2012/405, 2012. `http://eprint.iacr.org/`.
5. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10, 1988.
6. Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.
7. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *LNCS*, pages 192–206. Springer, 2008.
8. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.
9. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
10. Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *Proceedings of the 7th international conference on Security and cryptography for networks*, SCN'10, pages 182–199, Berlin, Heidelberg, 2010. Springer-Verlag.
11. Octavian Catrina and Sebastiaan De Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 134–150, Berlin, Heidelberg, 2010. Springer-Verlag.
12. Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *Proceedings of the 14th international conference on Financial Cryptography and Data Security*, FC'10, pages 35–50, Berlin, Heidelberg, 2010. Springer-Verlag.
13. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.
14. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.
15. Sharemind development team. The Sharemind framework. `http://sharemind.cyber.ee`, 2007.

16. Martin Franz and Stefan Katzenbeisser. Processing encrypted floating point signals. In *Proceedings of the thirteenth ACM multimedia workshop on Multimedia and security*, MM&Sec '11, pages 103–108, New York, NY, USA, 2011. ACM.
17. Roman Jagomägis. SecreC: a Privacy-Aware Programming Language with Applications in Data Mining. Master's thesis, Institute of Computer Science, University of Tartu, 2010.
18. Thomas Sean Kelso. Initial Analysis of Iridium 33-Cosmos 2251 Collision. Technical report, Center for Space Standards & Innovation, 2009.
19. Thomas Sean Kelso, David A. Vallado, Joseph Chan, and Bjorn Buckwalter. Improved Conjunction Analysis via Collaborative Space Situational Awareness. Technical report, 2008.
20. Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3rd edition, 1998.
21. Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In Proceedings of the 13th USENIX Security Symposium (2004), pp. 287-302., 2004.
22. NASA. Satellite Collision Leaves Significant Debris Clouds. *Orbital Debris Quarterly News*, 13(2):1–2, April 2009.
23. Stavros Paschalakis and Peter Lee. Double Precision Floating-Point Arithmetic on FPGAs. In *2003 IEEE International Conference on Field-Programmable Technology (FPT), 2003*, pages 352–358. IEEE, 2003.
24. Timothy P. Payne. First "confirmed" Natural Collision Between Two Cataloged Satellites. In *Proceedings of the Second European Conference on Space Debris*, volume ESA-SP 393, pages 597–600. ESA, 1997.
25. Boris A. Popov and Genadiy S. Tesler. *Вычисление функций на ЭВМ - справочник.* Naukova dumka, 1984.
26. RAND Corporation. Placeholder for the algorithm for conjunction analysis by RAND if they publish it. Technical report, RAND Corporation, 2013.
27. Adi Shamir. How to share a secret. *Commun. ACM*, 22:612–613, November 1979.
28. VIFF development team. The virtual ideal functionality framework. `http://viff.dk`, 2007.
29. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *Proc. of FOCS '82*, pages 160–164, 1982.