Point-Counting Method for Embarrassingly Parallel Evaluation in Secure Computation

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

[1] Cybernetica, Ülikooli 2, Tartu, Estonia
[2] Institute of Computer Science, University of Tartu, Liivi 2, Tartu, Estonia
[3] STACC, Ülikooli 2, Tartu, Estonia

Abstract. In this paper we propose an embarrassingly parallel method for use in secure computation. The method can be used for a special class of functions over fixed-point real numbers - namely, for functions f for which there exist functions g and h such that $g(f(x), x) = h(x)$ and $g(\cdot, x)$ is monotonous. These functions include $f(x) = \frac{1}{x}$, $f(x) = \sqrt{x}$ and $f(x) = \frac{1}{\sqrt{x}}$, but also any functions that can be represented as finding a root of a polynomial with secret coefficients and with a sufficiently low rank. The method can also be applied for other functions — such as the logarithm function. The method relies on counting techniques rather than evaluation of series, allowing the result to be obtained using less rounds of computations with the price of more communication in one round. Since the complexity of oblivious computing methods (like secret-shared multi-party computations (SMC)) is largely determined by the round complexity, this approach has a potential to give better performance/precision ratio compared to series-based approaches. We have implemented the method for several functions and benchmarked them using Sharemind SMC engine.

1 Introduction

The problem setting for privacy-preserving data mining (PPDM) is inherently self-contradictory. On one hand we collect data with a certain purpose, e.g. make policy decisions based on its analysis. On the other hand, due to increasing ease of data misuse, privacy regulations have become more strict in time. This in turn sets tighter limits to data collection, storage and utilisation.

In order to still make use of the data analysis, various PPDM techniques have been proposed. In this paper we will concentrate on secure multi-party computations (SMC), as this approach currently seems to offer good trade-off between performance loss (which is inevitable taking privacy restrictions into account) and provided security guarantees.

SMC applications work by sharing data between several computing nodes, whereas the privacy guarantees hold when not too many of them collude. Computations on such a set-up assume communication between these nodes. As the computing nodes should ideally be placed in physically and/or organisationally independent environments to provide the

⋆ This research was supported by the European Regional Development Fund through Centre of Excellence in Computer Science (EXCS) and the Estonian Research Council under Institutional Research Grant IUT27-1 and Estonian Doctoral School in Information and Communication Technologies

privacy guarantees, communication complexity of the computing protocols becomes a major bottleneck. Hence, development of the protocols having smaller communication complexity is essential to increase the performance of SMC applications.

This paper proposes a new approach for evaluating certain special-form functions in SMC setting. We will make use of the observation that communication complexity in SMC case is mostly determined by the number of communication rounds, whereas the amount of data sent within one round may be rather large without significantly decreasing the overall computation speed. [13, 3] So far this property was mostly useful while computing with large datasets, as many instances of the same protocols could run in parallel, giving a good aggregated speed. The methods described in this paper can fully use this property for small datasets, but are also flexible enough to be also well usable for large datasets.

One class of methods for function evaluation using many parallel independent attempts is know as Monte-Carlo methods. We can "throw" points into the plane to compute integrals, or onto the line to evaluate single-variable functions, and then count the points satisfying a certain relation. The key observation is that both "throwing" and counting can be implemented in parallel, allowing us to save considerably on communication rounds. In this paper we will not actually select the points randomly, hence the resulting method can be viewed as only inspired by Monte-Carlo approach.

The paper is organised as follows. We first describe the notation and the primitives that we can use in Section 3. Then Section 4 presents an important technical tool for the implementation of our method. In Section 5 we introduce the central algorithm, present its details and give the corresponding proofs for its correctness. In Section 6 we give examples of function to which the algorithm can be applied and also present how the algorithm can be used to compute logarithms or the roots of polynomials with secret coefficients. Finally, we give the benchmarking results in Section 7 and draw some conclusions in Section 8.

2 Previous Work

SMC has traditionally been done over integer values, however, in the recent years, there have been also solutions that use number types that represent real numbers, such as fixed-point numbers or floating-point numbers. Such data types allow for applications that are not possible when using only integer type such as the satellite collision problem [10], privacy-preserving statistical analysis [9], QR-decomposition of matrices [14], secure linear programming with applications in secure supply chain management [5, 11], and other problems.

Catrina and Saxena developed secure multiparty arithmetic on fixed-point numbers in [6], and their framework was extended with various computational primitives (like inversion and square root) in [6] and [14].

However, since the precision of fixed-point is inherently limited, secure floating-point arithmetic has been developed independently by various groups of researchers, see Aliasgari *et al.* [1], Liu *et al.* [15], and Kamm and Willemson [10].

In [12], we designed protocols that use fixed-point arithmetic inside of floating-point protocols for better performance.

However, since the fixed-point protocols we used in that paper used polynomial approximation to calculate different functions, the resulting accuracy was limited. This is because better approximation polynomials require computing greater powers of x and also have larger coefficients. These in turn increase the cumulative rounding errors resulting from the fixed-point format and thus, past some point, increasing the accuracy of the polynomial does not increase the accuracy of the method since the rounding error will dominate past that point. This also means that the method was not really flexible concerning the performance-precision trade-off. Also, the value of the polynomial might go outside a certain range and thus correction protocols must be applied to the fixed-point result.

3 Preliminaries

Our methods aim at working with fixed-point real numbers. The amount of bits after the radix point is globally fixed and is usually denoted by m. We will not introduce special notation for distinguishing between different types of numbers, as the type of a number will generally be clear from the context.

We differentiate between secret (private) and public values in this work. To denote that the value of some variable x is secret, we denote it with $[\![x]\!]$. A concrete instantiation of the value protection method depends on the implementation of the computation system. The reader may think of the value $[\![x]\!]$ as being secret shared [16], protected by fully homomorphic encryption [7, 8] or some other method allowing to compute with the protected values.

Given a set X and a function f we denote with $f(X)$ the set $\{f(x)|x \in X\}$. Given a number x, and a positive integer k we will define by $\beta_k(x)$ the number formed by the $n-k$ least significant bits of x where n is the total number of bits of our data type. We also define $\alpha_k(x) := x - \beta_k(x)$. In this paper, $\log x$ refers to the binary logarithm $\log_2(x)$. When we give a bit-decomposition of a value x by $x_0, x_1, \ldots, x_{n-1}$, we presume that x_{n-1} is the most significant bit.

3.1 Security Setting

We assume that we use a system that has some specific computational primitives and the universal composability property - that is, if two protocols are secure then their composition is also secure. More precisely, we will assume availability of the following computational primitives working on private fixed-point values.

- Addition
- Multiplication of public fixed-point numbers and private integers.

– Comparison. We assume access to an operator $\mathsf{LTEProtocol}([\![a]\!], [\![b]\!])$ that takes the value $[\![1]\!]$ if $a \leq b$, and $[\![0]\!]$ otherwise. We will also use notation $c = a \overset{?}{\leq} b$ to express this comparison operator.
– For functions f specified in the section 5, we must have some easily computable functions g and h so that $g(f(x), x) \equiv h(x)$ on private inputs. These g and h depend on which function f we want to compute, but to achieve efficient implementation they must be fast to evaluate on private fixed-point values. For example, for inverse and square root functions that we implemented, we only needed a small number of fixed-point multiplications to implement them.

All these universally composable primitives are available on several privacy-preserving computation frameworks, e.g. the framework by Catrina and Saxena [6] or Sharemind [2, 12].

4 Simple Example of the Main Idea

We start by presenting a simple method for evaluating functions where we know that the input belongs to some small range and where the function is suitably well-behaved. This method is similar in essence to the main method of the paper. We present this method here to help build intuition about the general strategy and thus make the main method easier to understand. Like the main method, this method uses many comparison operations in parallel to bring the round complexity to a minimum at the cost of increased number of computations in one round.

Often when we need to evaluate some function, we have some kind of information about it letting us know that it belongs to some small range $[a, b)$. If the function is twice derivable and its first two derivatives are not too large, we can use the following method. For a secret input $[\![x]\!]$ and a range $[a, b)$, we choose a large number of equidistributed points a_i where $a_i := a + i \cdot h$ for some small h. We then compute $[\![c_i]\!] := a_i \overset{?}{\leq} [\![x]\!]$. Then let $[\![d_0]\!] := 1 - [\![c_0]\!]$ and $[\![d_i]\!] := [\![c_{i-1}]\!] - [\![c_i]\!]$ for all $i \geq 1$. Note that only one of the variables $[\![d_i]\!]$ is equal to one, the rest are equal to zero. The j for which $[\![d_j]\!] = 1$ is the greatest j for which a_j is smaller than or equal to $[\![x]\!]$. Thus a_j is either the closest or second closest a_i to x and $f(a_j)$ can be considered as an approximation for $f([\![x]\!])$.

Thus we compute the scalar product $\sum_i f(a_i) \cdot [\![d_i]\!]$.

We noted that $\sum_i f(a_i) \cdot [\![d_i]\!] = f(a_j)$ where $a_j \leq [\![x]\!] < a_{j+1}$. We note that $|x - a_j| \leq h$. We assumed that f has first and second derivatives in $[a, b)$. Let $c_1 := \max_{y \in [a,b)} |f'(y)|$ and $c_2 := \max_{y \in [a,b)} |f''(y)|$. Then, according to Taylor's theorem, $|f(x) - f(a_j)| \leq c_1 h + \frac{c_2 h^2}{6}$. We can also add the error resulting from the inaccuracy of the fixed-point representation, but the error will be dominated by $c_1 h$.

Note that the method might also be usable for functions that do not have first and second derivatives, but in that case, a different error estimation is needed.

Data: $[\![x]\!], h, \ell, \{b_i\}_{i=0}^{\ell-1}, a, b$
Result: Given a secret number $[\![x]\!]$ such that $[\![x]\!] \in [a, b)$, and numbers $b_i \approx f(a + i \cdot h)$, computes a number $[\![z]\!]$ that is approximately equal to $f(x)$.
$\{a_i\}_{i=0}^{\ell-1} \leftarrow a + i \cdot h$
$\{[\![c_i]\!]\}_{i=0}^{\ell-1} \leftarrow \mathsf{LTEProtocol}(\{a_i\}_{i=0}^{\ell-1}, \{[\![x]\!]\}_{i=0}^{\ell-1})$
$[\![d_0]\!] \leftarrow 1 - [\![c_0]\!]$
for $i = 1, i < \ell, i++$ **do**
  $[\![d_i]\!] \leftarrow [\![c_{i-1}]\!] - [\![c_i]\!]$
end
$[\![z]\!] = \sum_{i=0}^{\ell-1} [\![d_i]\!] \cdot [\![b_i]\!]$
return $[\![z]\!]$

Algorithm 1: Simple example of the main idea

5 Functions with Simply Computable Monotone Inverses in Bounded Areas

There are functions that, in terms of elementary operations such as addition and multiplication, can be relatively complicated to compute but for which the inverse functions require much less computational power. For example, computing $\sqrt[k]{x}$ requires computing an approximation series and it is only accurate in a small interval, but computing x^k requires only approximately $\log k$ multiplications. Thus it would be useful if we could use computing x^k to compute $\sqrt[k]{x}$. Since round complexity is the important factor here, it is sufficient if we can compute $\sqrt[k]{x}$ by computing many instances of x^k in parallel.

We shall now describe a method that follows a similar idea. The main idea is the following. We want to compute a function $f(x)$ where the input x is secret, but of which we know that $f(x)$ belongs to the interval $[a, a + 2^k)$ where a can be private or public and where there are are easily computable functions g and h where $g(f(x), x) \equiv h(x)$ where $g(\cdot, x)$ is also monotonous in the area $[a, a + 2^k)$. For example, if $f(x) = \frac{1}{\sqrt{x}}$, then $g(x, y) = x^2 \cdot y$ and $h(x) = 1$.

Suppose that we know that the output $f(x)$ will be in some $[a, a + 2^k)$ and that we want to achieve precision 2^{k-s}. We then consider values $a_i := a + i \cdot 2^{k-s}$ for every $i \in \{1, \ldots, 2^s - 1\}$, compute $g(a_i, x)$, do secret comparisons by setting $c_i := g(a_i, x) \overset{?}{\leq} h(x)$ if g is increasing and $c_i := g(a_i, x) \overset{?}{\geq} h(x)$ if g is decreasing and finally set the result to be $a + 2^{k-s} \cdot (\sum c_i)$.

Intuitively, this gives a correct answer because for one $j \in \{0, \ldots, 2^s\}$, a_j is approximately equal to $f(x)$. If g is increasing, we can measure the position of this j in $\{0, \ldots, 2^s\}$ by testing whether $g(a_i, x) \overset{?}{\leq} h(x)$ for all i — due to monotonicity, for all i smaller than j, $g(a_i, x) \leq h(x)$ but for all i greater than j, $g(a_i, x) > h(x)$. Thus the number of i that "pass the test" is proportional to the position of j in $\{0, \ldots, 2^s\}$. A similar argument holds when g is decreasing.

The following theorem shows why the approach works.

Theorem 1. *Let f be a function. Let g, g_x and h be such functions that $g(f(x), x) \equiv h(x)$, $g_x(y) := g(y, x)$ and g_x is strictly monotonous in $[a, a + 2^k)$. Let $x \in X$ be such that $f(x) \in [a, a + 2^k)$. Let $y_0, y_1, \ldots, y_{2^s}$ be such that $y_i := a + i \cdot 2^{k-s}$. Let $Y := \{y_1, \ldots, y_{2^s-1}\}$. Let $Z := g_x(Y)$.*

Let $j := |\{y_i \in Y | g_x(y_i) \leq h(x)\}|$ and $j' := |\{y_i \in Y | g(y_i, x) \geq h(x)\}|$. Then $f(x) \in [y_j, y_{j+1})$ if g is monotonously increasing and $f(x) \in [y_{j'}, y_{j'+1})$ if it is monotonously decreasing.

Proof. We will give a proof for a monotonously increasing g_x. The proof for monotonously decreasing g_x is the same, *mutatis mutandis*. First note that $g_x(y_1) < g_x(y_2) < \cdots < g_x(y_{2^s-1})$, because g_x is monotonously increasing. We know that $f(x) \in [y_r, y_{r+1})$ for some r. Since g_x is monotone, this is equivalent to $g_x(f(x)) \in [g_x(y_r), g_x(y_{r+1}))$. Rewriting this gives us $h(x) \in [g_x(y_r), g_x(y_{r+1}))$. Because $g_x(y_1) < g_x(y_2) < \cdots < g_x(y_{2^s-1})$, this is equivalent to $h(x) \geq g_x(y_1), \ldots, g_x(y_r)$ and $h(x) < g_x(y_{r+1}), \ldots, g_x(y_{2^s-1})$, i.e. $|\{g(y_i) \in Z | h(x) \geq g(y_i)\}| = r$. This in turn gives us $|\{y_i \in Y | h(x) \geq g_x(y_i)\}| = r$, which gives us $r = j$ which is what we wanted to show. □

We shall use this theorem for Algorithm 2. Namely, to compute $f([\![x]\!])$, we first create values $[\![y_1]\!], [\![y_2]\!], \ldots, [\![y_{2^s-1}]\!]$ be such that $[\![y_i]\!] := [\![a]\!] + i \cdot 2^{k-s}$. We compute, $g([\![y_i]\!]) \overset{?}{\leq} h(x)$ for every i, in parallel. Then, using the LTEProtocol in parallel, we set c_i to be $g(y_i) \overset{?}{\leq} h(x)$ for every i. We compute $r = \sum c_i$. By the theorem, $r = j$ and thus we set the answer to be $[\![a]\!] + r \cdot 2^{k-s}$.

We shall now give two remarks about how the theorem still applies for some slightly weaker assumptions. We did not use these assumptions in the theorem due to the sake of clarity.

Remark 1. Note that even if $h(x)$ is not easily computable but there exists an easily computable function $\tilde{h}(x)$ such that $h(x) \in [g_x(y_r), g_x(y_{r+1})) \Leftrightarrow \tilde{h}(x) \in [g_x(y_r), g_x(y_{r+1}))$ then we can replace $h(x)$ with $\tilde{h}(x)$ in the algorithm and the output is the same.

Remark 2. We also note that it is not strictly necessary for g_x to be monotonous in $[a, a + 2^k)$. It suffices for it to be monotonous in $[a, a + 2^k - 2^{k-s}]$.

When we refer to functions g, g_x, h or $\tilde{h}$ later in this paper, we assume that they have the meanings described in this section. Also, when the function $g(x, y)$ does not depend on y, we shall just write $g(x)$. Also, in that case, $g_x(y) \equiv g(y)$ and thus we will write $g(y)$ instead of $g_x(y)$.

5.1 Iteration

Note that when the range of input is large and we want good accuracy, we have to perform a large number of tests in parallel — i.e. s will be rather large. However, due to network

Data: $[\![x]\!], [\![a]\!], k, n, s, \mathsf{sign}$
Result: Computes one round of function f of $[\![x]\!]$ where we already know that $f(x) \in [[\![a]\!], [\![a]\!] + 2^k)$. Here g and h are functions so that $g(f(x), x) \equiv h(x)$. The public flag sign is 1 if the function is increasing and 0 if it is decreasing. We use $2^s - 1$ test points and we work on n-bit data types.

$[\![w]\!] \leftarrow h([\![x]\!])$
$\{[\![a_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![a]\!] + i \cdot 2^{k-s}\}_{i=1}^{2^s-1}$
$\{[\![b_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![g(a_i, x)]\!]\}_{i=1}^{2^s-1}$
if sign $== 1$ **then**
  $\{[\![c_i]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![b_i]\!]\}_{i=1}^{2^s-1}, \{[\![w]\!]\}_{i=1}^{2^s-1})$
else
  $\{[\![c_i]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![w]\!]\}_{i=1}^{2^s-1}, \{[\![b_i]\!]\}_{i=1}^{2^s-1})$
end
$[\![c]\!] = \sum_{i=1}^{2^s-1} 2^{k-s} [\![c_i]\!]$
return $[\![a]\!] + [\![c]\!]$

Algorithm 2: Computing a function with an easily computable inverse.

saturation, there is an upper bound to how many operations does it make sense to perform in parallel. Let this number be 2^σ. Up to $2^{\sigma-1}$, doubling the number of tests should increase the overall computation time by a factor that is strictly smaller than 2. Past that point, however, doubling the number of tests will double the performance time. This is not a very serious loss, but we can also propose a method that can ideally achieve much greater accuracy gain while only doubling the performance time.

The idea is, in essence, 2^σ-ary search. We note that in the beginning of Algorithm 3 we start with the knowledge that $f([\![x]\!]) \in [[\![a]\!], [\![a]\!] + 2^k)$ and in the end we know that $f([\![x]\!]) \in [[\![a']\!], [\![a']\!] + 2^{k'})$ where k' is smaller than k. Thus it is rather natural to run Algorithm 3 again assuming that $f([\![x]\!]) \in [[\![a']\!], [\![a']\!] + 2^{k'})$ with a suitable number of test points in that interval. This can be done several times in a row, until the precision we want has been achieved.

More precisely, suppose that we want to compute v instances of some function f in parallel with accuracy of t bits and so that we beforehand know that $[\![f(x_i)]\!] \in [a_i, a_i + 2^k)$ for every $i \in \{1, \ldots, v\}$. Suppose also that our system can perform at most approximately 2^σ comparison operations or operations g in parallel and we want to achieve precision 2^t.

Then we have to perform approximately $\frac{(k-t)\log v}{\sigma}$ rounds where in every round the total number of operations performed is no greater than 2^σ.

However, if the number of operations we wish to do in parallel is too great, then we must perform more than 2^σ operations in one round. In this case, we shall compute only one extra bit each round because that is the smallest possible amount.

The resulting procedure is presented as Algorithm 3, where the *RoundF* subroutine refers to Algorithm 2.

Data: $v, \{[\![x_i]\!]\}_{i=0}^{v-1}, \{[\![a_i]\!]\}_{i=0}^{v-1}, k, \sigma, n, t$
Result: Computes function f of values $\{[\![x_i]\!]\}_{i=0}^{v-1}$. We know that $[\![f(x_i)]\!] \in [a_i, a_i + 2^k)$ for all $i \in \{0, \ldots, v-1\}$. We can perform at most 2^σ comparison or g operations in parallel. We work on n-bit data types and we wish to achieve precision 2^t

$s \leftarrow \max\{\lfloor \sigma - \log v \rfloor, 1\}$
$r \leftarrow \lfloor \frac{k-t}{s} \rfloor$
$s' \leftarrow k - t - s \cdot r$
if $s' == 0$ **then**
  $s' \leftarrow s$
  $r \leftarrow r - 1$
end
$\{[\![y_{0,i}]\!]\}_{i=0}^{v-1} \leftarrow RoundF(\{[\![x_i]\!]\}_{i=0}^{v-1}, \{[\![a_i]\!]\}_{i=0}^{v-1}, k, n, s')$
for $j = 1; j \leq r; j++$ **do**
  $\{[\![y_{j,i}]\!]\}_{i=0}^{v-1} \leftarrow RoundF(\{[\![x_i]\!]\}_{i=0}^{v-1}, \{[\![y_{j-1,i}]\!]\}_{i=0}^{v-1}, k - s' - (j-1)s, n, s)$
end
return $[\![y_r]\!]$

Algorithm 3: Computing f using iteration

6 Applications of the Method

The class of functions f described in Theorem 1 (for which there exist easily computable functions g and h such that $g(f(x), x) \equiv h(x)$ and $g(\cdot, x)$ is also monotonous) is rather abstract and not easy to interpret. This Section studies this class more closely.

The functions described by Theorem 1 are perhaps best understood when considering the possible options for the easily computable functions g, h and $\tilde{h}$. Which functions exactly are easy to compute depends on the underlying implementation of secure computation engine. Typically such functions would include constants, additions, subtractions, multiplications and their compositions. However, depending on the system, other operations such as bit decompositions, shifts or other functions might be cheap and thus different functions might belong into that class in that case.

The compositions of constants, additions, subtractions, multiplications are polynomials. Thus, one subset of the functions computable using this method are equivalent to finding the root of a polynomial with secret coefficients in a range where the polynomial is injective.

For example:

- computing $\frac{1}{[\![a]\!]}$ is equivalent to finding a root of $[\![a]\!]x - 1 = 0$;
- computing $\sqrt{[\![a]\!]}$ is equivalent to finding a root of $x^2 - [\![a]\!] = 0$;
- computing $\frac{1}{\sqrt{[\![a]\!]}}$ is equivalent to finding a root of $[\![a]\!]x^2 - 1 = 0$;
- computing $\frac{[\![a]\!]}{[\![b]\!]}$ is equivalent to finding a root of $[\![b]\!]x - [\![a]\!] = 0$.

This class of problems can also be extended to finding the roots of polynomials with secret coefficients in general, whether they are injective in an area or not, and this is done in Section 6.1. Later, in Section 6.2, we will present the computation routine for binary logarithm.

6.1 Finding Roots of Polynomials

We saw that finding the roots of injective polynomials is a large subclass of problems that can be solved using the point-counting method described above.

We will now present a method for making point-counting applicable also for polynomials that are not injective in the given interval. Denoting the rank of the polynomial by k, the extended method will possibly be up to k^2 times slower, hence it should be used only for polynomials with a suitably small rank.

The key observation for the extended method is the fact that we can still use the point-counting method if we can divide $[a, b)$ into intervals $[a, c_1), [c_1, c_2), .., [c_k, b)$ so that the polynomial $p(x)$ is monotone in all those intervals — we can then separately use the point-counting method in all those intervals.

The polynomial is monotone in an interval if the derivative of the polynomial does not change signs there. Since the derivative of a polynomial is a continuous function, it changes signs only when it is equal to zero. Thus we can find the points $c_1, \ldots, c_k$ by computing the roots of $p'(x)$. Now we again must find the roots of a polynomial — but that polynomial has a smaller rank than the original one. This, rather naturally, gives us a recursive algorithm. If $p'(x)$ is an injective function, we can directly use the point-counting algorithm. If it is not, we can compute its roots recursively.

If the rank of the polynomial $p(x)$ is k, then its $(k-1)$st derivative is a linear function and thus injective, which means that the recursion has no more than $k-1$ steps.

We presume that we have access to the following functions.

First, we naturally assume that we have access to the function that evaluates the polynomial. We denote with $\mathsf{p}([\![a_0]\!], \ldots, [\![a_n]\!], [\![x_0]\!])$ the function that evaluates the polynomial $\sum_{i=0}^{n} [\![a_i]\!] x^i$ at $[\![x_0]\!]$.

Second, we assume access to the function $\mathsf{Der}([\![a_0]\!], \ldots, [\![a_n]\!])$ that takes in the coefficients of a polynomial and returns the coefficients of its derivative.

Third, we presume that we have access to a version of Algorithm 3 that, in an interval where a polynomial is injective, returns a root of the polynomial if it has one or an endpoint of the interval if it does not. However, we need to modify the function *RoundF* that it calls, i.e. Algorithm 2. We shall replace it with the Algorithm 4 which differs from Algorithm 2 in two ways.

First, unlike in Algorithm 2, we do not know the length of the interval where our result may be. It might even happen that the interval has length zero. We solve this problem in the following way. Suppose that we know that our solution is in $[[\![a]\!], [\![b]\!]]$.

We then compute the values of g_x as usual in the interval $[[\![a]\!], [\![a+2^k]\!])$ where 2^k is such a number such that $[\![b]\!] \leq [\![a]\!]+2^k$. However, now we also compare every point $[\![a_i]\!] = [\![a]\!]+i \cdot h$ with $[\![b]\!]$. After computing the comparison vector $\{[\![c_i]\!]\}$, we compute $[\![c_i']\!] := [\![a_i]\!] \overset{?}{\leq} [\![b]\!]$ and compute $[\![c_i]\!] = [\![c_i]\!] \cdot [\![c_i']\!]$. We then proceed as usual. After this procedure we can be certain that the result is in $[[\![a]\!], [\![b]\!]]$.

Note that although we use intervals in the format $[a, b]$ instead of $[a, a + 2^t)$, we can still use the Theorem 1 due to Remark 2.

The second way Algorithm 4 differs from Algorithm 2 is the fact that we do not know whether the polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ is increasing or decreasing in the interval $[[\![a]\!], [\![b]\!]]$ where it is injective. We solve this problem by executing the algorithm in both cases and then computing $p(a) \overset{?}{\leq} p(b)$ to perform oblivious choice between the two options.

Because p is injective in the interval, the only case when it can happen that $p([\![a]\!]) = p([\![b]\!])$ is when $[\![a]\!] = [\![b]\!]$, but then the output of the function is always $[\![a]\!]$ and does not depend on whether we use the increasing or decreasing version of the algorithm. In other cases comparing $p([\![a]\!])$ and $p([\![b]\!])$ will give us whether the function is increasing or decreasing in that interval and thus the correct output. Thus we obtain Algorithm 4.

Data: $[\![x]\!], [\![a]\!], [\![b]\!], k, n, s$
Result: Computes one round of function f of $[\![x]\!]$ where we already know that $f(x) \in [[\![a]\!], [\![b]\!]]$. Here g and h are functions so that $g(f(x), x) \equiv h(x)$. We use $2^s - 1$ test points and we work on n-bit data types. We know that the function is monotone but not whether it is increasing or decreasing.

$[\![w]\!] \leftarrow h([\![x]\!])$
$\{[\![a_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![a]\!] + i \cdot 2^{k-s}\}_{i=1}^{2^s-1}$
$\{[\![b_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![g(a_i, x)]\!]\}_{i=1}^{2^s-1}$
$\{[\![c_{i,0}]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![b_i]\!]\}_{i=1}^{2^s-1}, \{[\![w]\!]\}_{i=1}^{2^s-1})$
$\{[\![c_{i,1}]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![w]\!]\}_{i=1}^{2^s-1}, \{[\![b_i]\!]\}_{i=1}^{2^s-1})$
$\{[\![c'_i]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![b_i]\!]\}_{i=1}^{2^s-1}, \{[\![b]\!]\}_{i=1}^{2^s-1})$
$\{[\![c_{i,0}]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![c_{i,0}]\!]\}_{i=1}^{2^s-1} \cdot \{[\![c'_i]\!]\}_{i=1}^{2^s-1}$
$[\![c_0]\!] = 2^{k-s} \cdot \sum_{i=1}^{2^s-1} [\![c_{i,0}]\!]$
$\{[\![c_{i,1}]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![c_{i,1}]\!]\}_{i=1}^{2^s-1} \cdot \{[\![c'_i]\!]\}_{i=1}^{2^s-1}$
$[\![c_1]\!] = 2^{k-s} \cdot \sum_{i=1}^{2^s-1} [\![c_{i,1}]\!]$
$[\![z_a]\!] \leftarrow \mathsf{p}([\![a_0]\!], \ldots, [\![a_n]\!], [\![a]\!])$
$[\![z_b]\!] \leftarrow \mathsf{p}([\![a_0]\!], \ldots, [\![a_n]\!], [\![b]\!])$
$[\![z]\!] \leftarrow \mathsf{LTEProtocol}(z_a, z_b)$
$[\![c]\!] \leftarrow [\![z]\!] \cdot [\![c_0]\!] + (1 - [\![z]\!]) \cdot [\![c_1]\!]$
return $[\![a]\!] + [\![c]\!]$

Algorithm 4: Computing a function with an easily computable inverse in a secret interval. Function may be either increasing or decreasing.

If we thus replace the call to Algorithm 2 with a call to Algorithm 4 in Algorithm 3, we shall obtain a function that we call $\mathsf{injecRoot}([\![a_0]\!], \ldots, [\![a_n]\!], [\![a]\!], [\![b]\!])$ that takes in a secret interval $[[\![a]\!], [\![b]\!]]$ and the secret coefficients $[\![a_0]\!], \ldots, [\![a_n]\!]$ of a polynomial such that the polynomial has at most one root in $[[\![a]\!], [\![b]\!]]$. The function outputs the root of the polynomial in $[[\![a]\!], [\![b]\!]]$ if it exists. If it does not exist, the function outputs the point $[\![a]\!]$ if the function has only positive values and is increasing in the interval or has only negative

values and is decreasing in the interval. In other cases, it outputs the largest representable value in $[[\![a]\!], [\![b]\!]]$.

We shall now present Algorithm 5 for the function $\mathsf{polyRoot}([\![a_0]\!], \ldots, [\![a_n]\!], [\![a]\!], [\![b]\!], t)$ that returns n values, in increasing order, among which are all the real roots of the polynomial.

First it finds the polynomial that is the derivative of the polynomial $\sum_{i=0}^{n} [\![a_i]\!] x^i$. If that is a linear function, it applies the function injecRoot to it to obtain its root if it has one. If the derivative has a higher order, it recursively calls polyRoot to obtain $n-1$ values $c_1, \ldots, c_{n-1}$, in increasing order, among which are all the real roots of the derivative.

We then set $[\![c_0]\!] = [\![a]\!]$ and $[\![c_n]\!] = [\![b]\!]$. We then apply the function injecRoot to the original polynomial in the intervals $[[\![c_i]\!]+2^t, [\![c_{i+1}]\!]]$ where 2^t is the precision of the function injecRoot. We return the outputs of injecRoot, ordered.

Data: $[\![a_0]\!], \ldots, [\![a_n]\!], [\![a]\!], [\![b]\!], t$
Result: Gets as input the polynomial with coefficients $[\![a_0]\!], \ldots, [\![a_n]\!]$ and an interval $[[\![a]\!], [\![b]\!]]$. Returns n values, in an increasing order, among which are all the real roots of the polynomial. Has precision 2^t.
if $n > 1$ **then**
  $[\![b_0]\!], \ldots, [\![b_{n-1}]\!] \leftarrow \mathsf{Der}([\![a_0]\!], \ldots, [\![a_n]\!])$
  $[\![c_1]\!], \ldots, [\![c_{n-1}]\!] \leftarrow \mathsf{polyRoot}([\![b_0]\!], \ldots, [\![b_{n-1}]\!], [\![a]\!], [\![b]\!], t)$
  $[\![c_0]\!] \leftarrow a$
  $[\![c_n]\!] \leftarrow b$
  for $i = 0, i < n, i++$ **do**
    $[\![d_i]\!] \leftarrow \mathsf{injecRoot}([\![a_0]\!], \ldots, [\![a_n]\!], [\![c_i + 2^t]\!], [\![c_{i+1}]\!])$
  end
  return $[\![d_0]\!], \ldots, [\![d_{n-1}]\!]$
else
  return $\mathsf{injecRoot}([\![a_0]\!], \ldots, [\![a_1]\!], [\![a]\!], [\![b]\!])$
end

Algorithm 5: Computing roots of a polynomial

Based on Theorem 1, we note that each step gives correct answers provided that the function under question is injective in the intervals where it is called. Based on properties of the derivative of a polynomial we conclude the correctness of the algorithm.

The reason why we chose the specific intervals for injecRoot as $[[\![c_i]\!] + 2^t, [\![c_{i+1}]\!]]$ is the following. We know that the derivative of the function may be zero in $[[\![c_i]\!], [\![c_i]\!] + 2^t)$ and $[[\![c_{i+1}]\!], [\![c_{i+1}]\!] + 2^t)$, but not between those intervals. Thus the function is injective in $[[\![c_i]\!] + 2^t, [\![c_{i+1}]\!]]$ and the algorithm gives the desired output. Also, this does not exclude any points, given the precision level t.

6.2 Logarithm

In this Section we show how the point-counting method can be applied to computing binary logarithms.

As noted by Aliasgari *et al.* [1], an approximation of the exponential function can be computed using the bits of the input to obliviously choose between 2^{2^i} and 1 and then computing the product over all bits — we can use this for the function g. At first, it may seem that this requires us to perform bit-decomposition and many multiplications for computing the function g. However, we will later see that it can be done in a manner where we only need a multiplication of a private and a public value to compute g.

To avoid technical details arising from the need to handle negative numbers, we only present here the point-counting method for logarithms that only works on inputs greater than one. We refer to the discussion in Section 7 how to overcome this limitation.

Let us have input $[\![x]\!]$ and suppose that we want to compute $[\![\log x]\!]$. We assume that $[\![\log x]\!] \in [[\![a]\!]2^b, ([\![a]\!]+1)2^b)$ for some $[\![a]\!]$ and b. Let us have n-bit numbers as input. Let us also have computed the value $[\![2^{2^b a}]\!]$. We let $f(x) = \log x$ and $g(y) = \overline{g_{b,s}}(y_0, \ldots, y_{n-1})$, where $y_0, \ldots y_{n-1}$ are the bits of y and $\overline{g_{b,s}}(y_0, \ldots, y_{n-1})$ is the function defined in the following way.

$$\overline{g_{b,s}}(y_0, \ldots, y_{n-1}) = 2^{2^b a} \Pi_{i=n-b}^{s}(y_i \cdot 2^{2^i} + (1 - y_i) \cdot 1).$$

Note that this performs oblivious choice between 2^{2^i} and 1 every step using the bit y_i, and essentially computes an approximation of the exponent that uses only the first s bits of y. In fact, $g(y) \equiv 2^{\alpha_s(y)}$. Let us have 2^{s-b} test points a_r in such a way that $a_i = i \cdot 2^{n-s}$. Thus $h(x) = \overline{g_{b,s}}(\log x) = 2^{\alpha_s(\log x)}$.

This gives us $h(x) = 2^{\alpha_s(\log x)}$. However, this is difficult to compute.

We now use Remark 1 and set $\tilde{h}(x) = x$. We need to confirm that $h(x) \in [g(a_r), g(a_{r+1})) \Leftrightarrow \tilde{h}(x) \in [g(a_r), g(a_{r+1}))$ Indeed,

$$h(x) \in [g(a_r), g(a_{r+1})) \Leftrightarrow 2^{\alpha_s(\log x)} \in [2^{\alpha_s(a_r)}, 2^{\alpha_s(a_{r+1})}) \Leftrightarrow \alpha_s(\log x) \in [\alpha_s(a_r), \alpha_s(a_{r+1})) \Leftrightarrow$$

$$\alpha_s(\log x) = \alpha_s(a_r) \Leftrightarrow x = 2^{\alpha_s(a_r)} \cdot 2^{\beta_s(x)} \Leftrightarrow x \in [2^{\alpha_s(a_r)}, 2^{\alpha_s(a_{r+1})}) \Leftrightarrow \tilde{h}(c) \in [g(a_r), g(a_{r+1})).$$

The equivalence $\alpha_s(\log x) \in [\alpha_s(a_r), \alpha_s(a_{r+1})) \Leftrightarrow \alpha_s(\log x) = \alpha_s(a_r)$ holds because the only point in $[\alpha_s(a_r), \alpha_s(a_{r+1}))$ to which $\alpha_s(\cdot)$ maps is $\alpha_s(a_r)$.

We now note that if we know $[\![2^{2^b a}]\!]$, then computing $g(x)$ can be done at the cost of multiplying a public value with a private value. Namely, we note that given the bit representation of a_i as $a_{i_{n-1}} a_{i_{n-2}} \ldots a_{i_0}$, the product $\Pi_{j=n-b}^{s}(a_{i_j} \cdot 2^{2^j} + (1 - a_{i_j}) \cdot 1)$ is public since the individual bits $a_{i_{n-b}}, \ldots, a_{i_s}$ are public as they only depend on i. Thus we only need to compute the product of $[\![2^{2^b a}]\!]$ and $\Pi_{j=n-b}^{s}(a_{i_j} \cdot 2^{2^j} + (1 - a_{i_j}) \cdot 1)$ to compute $g(a_i)$.

We also note that the constraint that we should have the value $[\![2^{2^b a}]\!]$ before computation is not too restricting. If we have performed no rounds before, then it can be set to 1.

However, if it is not the first round, then we can compute it based on the values we obtained from the previous round using the method described in Section 4. Note that we have computed values $[\![c_i]\!] = [\![g(x)]\!] \stackrel{?}{\leq} [\![x]\!]$. We then compute $[\![d_0]\!] := 1 - [\![c_0]\!]$ and $[\![d_i]\!] := [\![c_{i-1}]\!] - [\![c_i]\!]$ for all $i \in \{1, \ldots, 2^s - 1\}$. We now note that there is only one $[\![j]\!]$ for which $[\![d_j]\!] = 1$, namely, the one for which $g(a_{[\![j]\!]}) \leq [\![x]\!] < g(a_{[\![j+1]\!]})$. Now we compute $\sum_{i=0}^{2^s-1} d_i [\![g(a_i)]\!]$ which is equal to the $g(a_{[\![j]\!]})$ for which it holds that $g(a_{[\![j]\!]}) \leq [\![x]\!] < g(a_{[\![j+1]\!]})$. Note that this means $2^{\alpha_s(a_{[\![j]\!]})} \leq [\![x]\!] < 2^{\alpha_s(a_{[\![j]\!]})+2^{n-s}}$. We note that by taking $b = n - s$ and $a = \frac{\alpha_s(a_j)}{2^{n-s}}$ this is equivalent to $[\![a]\!]2^b \leq x < ([\![a]\!]+1)2^b$ and thus we can take $[\![g(a_j)]\!] = \sum_{i=0}^{2^s-1} d_i [\![g(a_i)]\!]$ for the new $[\![a]\!]2^b$.

7 Results

7.1 Security Settings of the Benchmarking Process

For implementation and benchmarking we used the Sharemind 3 SMC platform[4]. Sharemind 3 is based on secret sharing and is information-theoretically secure in the passive security model and contains the necessary universally composable primitives for fixed-point numbers, as described in section 3.1.

7.2 Benchmarks

We implemented four functions – inverse, square root and logarithm using the point-counting technique, and the Gaussian error function using the method described in Section 4. All three computing nodes for the Sharemind platform that we used contained two Intel X5670 2.93 GHz CPUs with 6 cores and had 48 GB of RAM. Although we optimised the methods concerning round-efficiency, total communication cost became the deciding factor for efficiency. Since these methods were designed to fully use the communication capacity of channels, communication cost is proportional to time. We performed tests for 32-bit numbers and 64-bit numbers as the basic integer data type, and with different precision levels. To see how vector size affects the performance, we ran tests for different vector sizes. We performed 20 tests for every value given here for the inverse, square root and the Gaussian error function and 10 tests for every value for the logarithm and averaged the result.

For clearer comparison, we used the floating-point methods for the inverse function and the square root function presented in [12], but replaced the fixed-point subroutines with the methods proposed in this paper. We also did not need as much correction algorithms, since our new algorithms allow for much higher accuracy without significant loss in performance. The results of the tests are presented in Tables 1, 2 and 4, respectively.

[4] http://sharemind.cyber.ee/

We chose the precision parameters for square root and inverse for the following reasons. We wished to have a near-maximal precision for both 32-bit and 64-bit numbers but for practical reasons, we implemented the method for 30 or 62 bits of precision. This precision is much higher than the previous results of [12] and [10] for the respective data types while taking about 2 to 6 times more time than the results of [12]. We also ran tests for 16 bits of precision for 32-bit numbers and 32 bits of precision for 64-bit numbers. These precisions are approximately half of the near-maximal precision but also close to the precisions of the protocols of [12].

Comparing these benchmarks with the near-maximal precision we can see how doubling the number of bits for precision also approximately doubles the execution time. Based on the nature of the protocol, we can also assume that this pattern also holds more generally — when a protocol with n bits precision would take time t, then the same protocol would take time $2t$ for $2n$ bits of precision. Consequently, this protocol can be used with reasonable efficiency for applications needing very high precision. Verifying this conclusion assumes implementation of our algorithms on a platform providing 128-bit primitive types, so this remains the subject for future research.

We can also see that for 64-bit numbers, the performance for vector size 10000 is poorer than for vector size 1000. This happens because for 64-bit numbers, $2^\sigma < 10000$ and thus we have to perform more operations in a round than can be computed in parallel.

Since precision is an important aspect of our methods, we included precision of other methods to those methods where the precision could be found or computed. For the error function, we can see that for small vector sizes, our implemented method outperforms previous methods for several different precisions, being able to be both faster and more precise than our previous result [12]. However, if the vector size is larger, then the performance quickly decreases as precision grows. Thus it is preferable to use this method only for small vector sizes.

As for the logarithm, we implemented a logarithm function for 32-bit and 64-bit fixed-point numbers. In the implementation presented in this paper we assumed that the result of the logarithm operator would be positive. However, it is straightforward to also implement the absolute value of the negative case using the same method and then execute one oblivious choice. This would increase the computation time by approximately two times.

Our implemented methods had precisions of 2^{-15} and 2^{-31}, respectively, due to the respective radix-point used — for the radix-point used, higher precision was not possible. We can see that while the method proposed by Aliasgari *et al* can achieve very high precision due to the nature of the floating-point data type, our method is faster, and for larger vector sizes, faster by several orders of magnitude.

8 Conclusions

This paper presented a method for oblivious evaluation of special-format single-variable functions, including, but not limited to functions that can be represented as finding roots

	1	10	100	1000	10000
Catrina, Dragulin,AppDiv2m 128 bits, [4]	3.39				
Catrina, Dragulin, Div2m, 128 bits [4]	1.26				
Kamm and Willemson, 32 bits,accuracy 2^{-18} [10]	0.17	1.7	15.3	55.2	66.4
Kamm and Willemson, 64 bits,accuracy 2^{-18} [10]	0.16	1.5	11.1	29.5	47.2
Krips and Willemson, 32 bits, accuracy 2^{-13} [12]	0.99	8.22	89.73	400.51	400.51
Krips and Willemson, 64 bits, accuracy 2^{-26} [12]	0.82	8.08	62.17	130.35	130.35
Current paper, 32 bits, accuracy 2^{-30}	0.34	3.60	21.97	98.13	106.15
Current paper 64 bits, accuracy 2^{-62}	0.24	2.21	10.8	48.1	37.1
Current paper, 32 bits, accuracy 2^{-16}	0.61	8.25	40.8	190.9	212.01
Current paper 64 bits, accuracy 2^{-32}	0.45	4.11	23.31	100.0	67.13

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

	1	10	100	1000	10000
Liedel, 110 bits, accuracy 2^{-78} [14]	0.204				
Kamm and Willemson, 32 bits [10]	0.09	0.85	7	24	32
Kamm and Willemson, 64 bits [10]	0.08	0.76	4.6	9.7	10.4
Krips and Willemson, 32 bits, accuracy 2^{-17} [12]	0.77	7.55	70.7	439.17	580.81
Krips and Willemson, 64 bits, accuracy 2^{-34} [12]	0.65	6.32	41.75	78.25	119.99
Current paper, 32 bits, accuracy 2^{-30}	0.30	2.98	19.97	94.13	101.8
Current paper 64 bits, accuracy 2^{-62}	0.21	1.93	9.23	44.79	37.13
Current paper, 32 bits, accuracy 2^{-16}	0.49	5.98	35.2	152.0	202.3
Current paper 64 bits, accuracy 2^{-32}	0.38	3.59	21.2	86.0	79.5

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

of polynomials with secret coefficients. Several important functions belong to this class (e.g. various power functions and binary logarithm). Our method is easy to implement and rather flexible as it can be used for various vector sizes and precisions, is designed to fully use the communication capacities of channels, and it offers good performance/precision ratio and can effectively be used for both small and large datasets and give maximal precision for fixed-point data types.

References

1. Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS*, 2013.
2. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In Sushil Jajodia and Javier Lopez, editors, *ESORICS'08*, volume 5283 of *LNCS*, pages 192–206. Springer Berlin / Heidelberg, 2008.
3. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

	1	10	100
Kamm and Willemson, 32 bits [10]	0.1	0.97	8.4
Kamm and Willemson, 64 bits [10]	0.09	0.89	5.8
Krips and Willemson, 32-bit, accuracy 2^{-13} [12]	0.5	4.41	30.65
Krips and Willemson, 64-bit, accuracy 2^{-13} [12]	0.46	4.13	21.97
Current paper, 32-bit, accuracy 2^{-10}	0.93	8.49	35.3
Current paper, 64-bit, accuracy 2^{-10}	0.86	7.01	22.36
Current paper, 32-bit, accuracy 2^{-11}	0.92	7.81	26.46
Current paper, 64-bit, accuracy 2^{-11}	0.84	5.93	13.55
Current paper, 32-bit, accuracy 2^{-12}	0.9	5.85	13.26
Current paper, 64-bit, accuracy 2^{-12}	0.81	3.94	6.13
Current paper, 32-bit, accuracy 2^{-13}	0.86	4.88	8.61
Current paper, 64-bit, accuracy 2^{-13}	0.75	2.59	3.75
Current paper, 32-bit, accuracy 2^{-14}	0.8	3.4	5.34
Current paper, 64-bit, accuracy 2^{-14}	0.57	1.19	1.54

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

	1	10	100	1000	10000	100000
Aliasgari, accuracy 2^{-256} [1]		12.36	12.5	13.3	13.3	13.5
Current paper, 32-bit, accuracy 2^{-15}	2.39	15.43	119.2	549.9	1023.6	1288.9
Current paper, 64-bit, accuracy 2^{-31}	0.90	6.8	37.9	152.5	244.3	275.6

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

4. Octavian Catrina and Claudiu Dragulin. Multiparty computation of fixed-point multiplication and reciprocal. In *Database and Expert Systems Application, 2009. DEXA '09. 20th International Workshop on*, pages 107–111, 2009.
5. Octavian Catrina and Sebastiaan Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security – ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 134–150. Springer Berlin Heidelberg, 2010.
6. Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *Financial Cryptography and Data Security*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer Berlin Heidelberg, 2010.
7. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09*, pages 169–178, 2009.
8. Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer Berlin Heidelberg, 2011.
9. Liina Kamm. *Privacy-preserving statistical analysis using secure multi-party computation*. PhD thesis, University of Tartu, 2015.
10. Liina Kamm and Jan Willemson. Secure floating-point arithmetic and private satellite collision analysis. Cryptology ePrint Archive, Report 2013/850, 2013. http://eprint.iacr.org/.
11. F. Kerschbaum, A. Schroepfer, A. Zilli, R. Pibernik, O. Catrina, S. de Hoogh, B. Schoenmakers, S. Cimato, and E. Damiani. Secure collaborative supply-chain management. *Computer*, 44(9):38–43, 2011.
12. Toomas Krips and Jan Willemson. Hybrid Model of Fixed and Floating Point Numbers in Secure Multiparty Computations. In *Proceedings of ISC 2014*, volume 8783 of *LNCS*, pages 179–197. Springer, 2014.
13. Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *ISC '11*, volume 7001 of *LNCS*, pages 262–277, 2011.
14. Manuel Liedel. Secure distributed computation of the square root and applications. In MarkD. Ryan, Ben Smyth, and Guilin Wang, editors, *Information Security Practice and Experience*, volume 7232 of *Lecture Notes in Computer Science*, pages 277–288. Springer Berlin Heidelberg, 2012.
15. Y.-C. Liu, Y.-T. Chiang, T. s. Hsu, C.-J. Liau, and D.-W. Wang. Floating point arithmetic protocols for constructing secure data analysis application.
16. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.