Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 22 (2013) 152 – 161

Procedia Computer Science

17th International Conference in Knowledge Based and Intelligent Information and Engineering Systems - KES2013

Floating Point Arithmetic Protocols for Constructing Secure Data Analysis Application

Yun-Ching Liu[a], Yi-Ting Chiang[a], Tsan-Sheng Hsu[a], Churn-Jung Liau[a], Da-Wei Wang[a,*]

[a]*Institute of Information Science, Academia Sinica, 128 Academia Road, Section 2, Nankang, Taipei 115, Taiwan*

Abstract

A large variety of data mining and machine learning techniques are applied to a wide range of applications today. Therefore, there is a real need to develop technologies that allows data analysis while preserving the confidentiality of the data. Secure multi-party computation (SMC) protocols allows participants to cooperate on various computations while retaining the privacy of their own input data, which is an ideal solution to this issue. Although there is a number of frameworks developed in SMC to meet this challenge, but they are either tailored to perform only on specific tasks or provide very limited precision. In this paper, we have developed protocols for floating point arithmetic based on secure scalar product protocols, which is required in many real world applications. Our protocols follow most of the IEEE-754 standard, supporting the four fundamental arithmetic operations, namely addition, subtraction, multiplication, and division. We will demonstrate the practicality of these protocols through performing various statistical calculations that is widely used in most data analysis tasks. Our experiments show the performance of our framework is both practical and promising.

© 2013 The Authors. Published by Elsevier B.V. Open access under CC BY-NC-ND license.

Selection and peer-review under responsibility of KES International

Keywords: Secure Multi-party Computations; Secure Scalar Product; Secure Floating Point Computation; Linear Regression;

1. Introduction

The fast development of today's web technology has been matched by the rapid growth of data distributed over the Internet. The performance of joint computation across several distributed databases is needed in a wide array of applications. As confidentiality of the data of the participants is also a prominent factor and often a vital consideration, secure multi-party computation (SMC) offers a means for participants to cooperate in accomplishing various computational tasks without compromising their privacy.

Many applications in scientific or business domain, such as regression and optimization, may require computation on real numbers of a fairly diverse range. A number of general frameworks for SMC have been proposed[3, 12, 14], but most of which can only perform integer-based computation. Although frameworks based on integers

*Corresponding author. Tel.: +886-2-27883799 ; fax: +886-2-27824814.
E-mail address: wdw@iis.sinica.edu.tw.

1877-0509 © 2013 The Authors. Published by Elsevier B.V. Open access under CC BY-NC-ND license.
Selection and peer-review under responsibility of KES International
doi:10.1016/j.procs.2013.09.091

can simulate fixed precision rational number computations, the range is limited and the rounding error accumulates rapidly after a large number of operations. Therefore, there is a real need for constructs that can perform non-integer computations with acceptable precision and performance.

In this paper, we will develop a set of SMC protocols for performing IEEE-754 floating point arithmetic operations, by utilizing a set of secure integer computation protocols based on secure scalar product protocols. Our current implementation assumes two parties, with an neutral third party as a commodity server. Although, it should be noted that our construct can be generalized to apply to more than two parties. The performance and practicality of our proposed protocols will be demonstrated through various analysis and measurements in real network environments, while performing a number of practical computations.

The organization of this paper is composed of six sections. Related works are reviewed in Section 2. And in Section 3, the data sharing scheme, secure scalar product protocol, the integer protocol suite, and the floating point point representation format are introduced. The specifications and implementation of the floating point arithmetic protocols, along with an analysis of their complexity are given in Section 4. Section 5 will demonstrate the practicality and performance of our protocols in the various experiments. Finally, we will give a brief conclusion and point out directions for future work in Section 6.

2. Related Works

First introduced by Yao[21] and extended by Goldreich et al.[11], there has been an on-going development in the field of multi-party secure computation. Among various constructs, integer-based secure scalar products has been found to be very practical in real applications.

Many constructs for multi-party secure computation based on secure scalar product has been proposed over the years, among which the performance of secure scalar product protocols based on the commodity model has been demonstrated to be extraordinarily efficient[19].

The theoretical aspects of the secure scalar product protocols have also been extensively investigated. A privacy measurement based on information theory proposed by Chiang et al.[5], proved that the invertible-matrix approach leaks at least half of the private information, whereas the commodity-based approach is perfectly secure. Wang et al. [17] have proven that no information-theoretically secure two-party protocol exists for scalar products without the use of a third party server, and Shen et al.[16] conducted an exploratory verification of the closure property of the commodity-based approach.

Whereas secure multi-party computation protocols are traditionally developed for integer computations, recent research developments have extended the horizon of SMC protocols to non-integer values. Protocols by Fouque et al.[7] handles computation of rational numbers. Catrina et al. [4] proposed protocols for real value computation using fixed point representation, while the protocols developed by Deiseroth et al. [8] are based on logarithmic representation. A set of protocols, based on homomorphic encryption and garbled circuits, for single precision floating point representation is proposed by Franz et al.[9], but only the specification of the construct are given, and no experiment were conducted. Recently, Aliasgari et al. [2] proposed another set of protocols for single precision floating point computation, based on a modification of fixed point number protocols.

However, according to the error estimations method in [13], it is easy to estimate the relative error of single precision floating point numbers in certain applications. For example, the relative error of the computation of the mean of 1000 single precision floating point numbers can reach as high as around 10^{-4}. Therefore, there is a real need for protocols that can achieve higher precisions, such as double precision floating point numbers.

3. Preliminaries

In this section, we first introduce the secret sharing scheme that our protocols are based on, and give a brief description of the secure scalar product protocol. The next subsection will be an introduction to the integer computation protocol suite, which will serve as the building block of our floating point based protocols. After this, we detail the specification of the floating point number format supported by our protocols, and finally the specification of a sharing scheme for floating point numbers by using three integers.

3.1. Additive Secret Sharing Scheme

Our protocols are based on additive secret sharing over $\mathbb{Z}_N$, that is a secret value x is split into n shares $x_1, x_2, ..., x_n \in \mathbb{Z}_N$to n parties, such that $x = \sum_{i=1}^{n} x_i$, and any $n-1$ subset $\{x_{i_1}, ..., x_{i_{n-1}}\}$ is uniformly distributed. The original secret can only be recovered, if and only if all the shares are combined together. Suppose $[a], [b]$ denote two values that are additively shared and c is a public constant. Since additive secret sharing is also a linear secret sharing scheme, addition $[a] + [b]$ and constant multiplication $c[a]$ can be performed by each party locally.

In this paper we mainly consider two party secure computation. Therefore, a secret value $x \in \mathbb{Z}_N$ is distributed in to two shares $x_1, x_2 \in \mathbb{Z}_N$, such that $x = (x_1 + x_2) \bmod N$, where x_1 is privately held by one party, and x_2 is privately held by the other.

3.2. Scalar Product Protocols

For two-party secure computation, suppose the two parties hold private inputs X_1 and X_2, respectively. After the execution of some protocol, they hold private outputs Y_1 and Y_2. The subscript of a variable denotes the party which owns the variable. We use $(X_1, X_2)\{plist\} \mapsto (Y_1, Y_2)$ to specify the corresponding protocols, where *plist* is a list of public variables. The formulation for the scalar product protocol follow's Goldreich's principle[10].

Specification 1 (Scalar-Product)**.** *Party 1 holds a private vector* $(x[1]_1, \ldots, x[d]_1)$ *and Party 2 hold a private vector* $(x[1]_2, \ldots, x[d]_2)$*, where* $x[i]_j$ *denotes the ith element of Party j's private vector. Both parties want to collaboratively execute the secure protocol*

$$((x[1]_1, \ldots, x[d]_1), (x[1]_2, \ldots, x[d]_2)) \mapsto (y_1, y_2)$$

such that $y_1 + y_2 = (\sum_{i=1}^{d} x[i]_1 \cdot x[i]_2)$ *mod N, where* y_j *is held by Party j, and that* $x[i]_1, x[i]_2, y_1, y_2 \in \mathbb{Z}_N$*, for each* $i = 1, \ldots, d$*.*

It can be observed from the specification that the output of the scalar product protocols are the additive sharing of the scalar product of the two private vectors held by the two parties.

The specific secure scalar product protocol construct we have adopted for our implementation is the commodity-based approach [6]. Although a third neutral party is needed for two party computation, this approach has extraordinarily efficient performance compared to several other constructs of secure scalar product protocols [19]. The protocol has been proven to be information theoretically secure against semi-honest adversaries [5][17].

3.3. The Integer Computation Protocol Suite

Since the input and output of secure scalar protocols are both additively shared, the outputs of previous protocols can be inputs to the next secure scalar product protocol. Therefore, it is possible to implement a protocol π composed of scalar product protocols. Since the input and output of each composing secure scalar product protocol, which are actually the intermediate results of π, are all additively shared, hence no intermediate results are revealed.

An integer computation protocol suite, developed by using the scalar product protocol as building blocks, is capable of performing various commonly-used operations [15][18]. A brief description of the specific protocols that are used in implementing the floating point protocols is given in Table 1.

It should be noted that the protocols in the integer computation protocol suite and the proposed floating-point protocols can be constructed from any scalar product protocol construct that follows specification 1, and can be easily extended to multi-party settings by decomposing various computations into pairwise-tasks between parties.

3.4. Our Floating Point Number Format

A floating point number consists of three parts, namely the sign bit, the exponent, and the mantissa. Let ℓ_s, ℓ_e, and ℓ_m, denote the length of the sign bit, the exponent field and the mantissa field in bits, respectively, and let $S = 2^{\ell_s}$, $E = 2^{\ell_e}$, $M = 2^{\ell_m}$. Hence the domains are $\mathbb{Z}_S$, $\mathbb{Z}_E$, and $\mathbb{Z}_M$, respectively. The bias of the exponent is defined as $bias = (2^{\ell_e - 1}) - 1$.

The Integer Computation Protocols		
Domain	Protocol	Description of output ($w_1 + w_2$)
$\mathbb{Z}_N$	Z_2-to-$Z_n(x_i, \mathbb{Z}_N)$	transform (x_1+x_2) from a string with elements shared in $\mathbb{Z}_2$ to an integer in $\mathbb{Z}_N$
$\mathbb{Z}_2$	Z_n-to-$Z_2(x_i, \mathbb{Z}_N)$	transform $(x_1 + x_2)$ from an integer in $\mathbb{Z}_N$ to a string with elements shared in $\mathbb{Z}_2$
$\mathbb{Z}_N$	Addition(x_i,y_i)	returns $(x_1 + x_2) + (y_1 + y_2)$
$\mathbb{Z}_N$	Product(x_i,y_i)	returns $(x_1 + x_2) \cdot (y_1 + y_2)$
$\mathbb{Z}_N$	If-Then-Else(d_i, x_i, y_i)	if $(d_1 + d_2) = 1$ then return $(x_1 + x_2)$ else return $(y_1 + y_2)$
$\mathbb{Z}_N$	Comparison(x_i)	if $(x_1 + x_2) < 0$ then return 1 else return 0
$\mathbb{Z}_N$	Equal-Zero(x_i)	if $(x_1 + x_2) = 0$ then return 1 else return 0
$\mathbb{Z}_N$	Shift-Right(x_i, n_i)	returns $(x_1 + x_2)$ shifted $(n_1 + n_2)$ digits to the right
$\mathbb{Z}_N$	Shift-Left(x_i,n_i)	returns $(x_1 + x_2)$ shifted $(n_1 + n_2)$ digits to the left
$\mathbb{Z}_2$	Or(x_i, y_i)	returns $((x_1 + x_2)$ OR $(y_1 + y_2))$
$\mathbb{Z}_2$	And(x_i, y_i)	returns $((x_1 + x_2)$ AND $(y_1 + y_2))$
$\mathbb{Z}_2$	Not(x_i)	returns (NOT$((x_1 + x_2))$)

Table 1: The integer protocols that are used in the floating point protocols. The integers x_i, y_i, d_i, and w_i are held by each party, where $i = 1$ denotes the integer is held by Party 1 and $i = 2$ denotes the integer held by Party 2 .

A widely used floating point number format is the IEEE-754 standard[1], which for single precision $\ell_s = 1, \ell_e = 8, \ell_m = 24$, with $bias = 127$ and for double precision $\ell_s = 1$, $\ell_e = 11$, $\ell_m = 53$, with $bias = 1023$. The mantissa in the standard is a fixed-point number, where the integer part is assumed to be a hidden-bit. Since the arithmetic rules for integer and fixed-point numbers are essentially the same, the difference only lies in the interpretation of the number, and as such we can loosely consider them interchangeable with integers.

The floating point number format that our protocol supports mostly follow the double precision of the IEEE-754 standard with the following minor exceptions:

- For ease of implementation, the hidden-bit in the IEEE-754 standard is made explicit and an extra bit is extended, thus making the integer part of the mantissa be represented by two bits. Hence the mantissa field length in our implementation is $\ell_m = 55$.
- Apart from the value zero, all other special values, such as "Not a number"(NaN), and exception handling, such as division by zero, are not supported, since if supported, they might entail some information leakage during the computation. When these conditions are encountered, the protocols compute as normal, but may produce erroneous results.
- The only rounding mode supported is truncation, while IEEE-754 also defines round to zero, round to $\pm\infty$, and round-to-nearest.

3.5. Componentwise Additive Sharing of Floating Point Numbers

In subsequent discussions, the symbols "$+_{fp}$", "$-_{fp}$", "$\times_{fp}$", and "$\div_{fp}$" denotes the normal floating point addition, subtraction, multiplication, and division, respectively.

The most natural way do define the additive secret sharing of a floating point number f is by $f = f_1 +_{fp} f_2$, where floating point numbers f_1 and f_2 are privately and separately held by the two parties. Since f_1 and f_2 a both uniformly distributed, the original secret f cannot only be recovered by combining both shares, which is comparable to that of integer additive secret sharing.

A floating point number f can also be represented as a 3-tuple of integers (s_f, e_f, m_f), where s_f is the sign, e_f is the exponent, and m_f is the mantissa. Hence, sharing a floating point number can be viewed as the sharing of these three integers.

Definition 3.1. *For a pair of 3-tuple of integers $t_1 = (s_{t_1}, e_{t_1}, m_{t_1})$, $t_2 = (s_{t_2}, e_{t_2}, m_{t_2})$, where $s_{t_i} \in \mathbb{Z}_S$, $e_{t_i} \in \mathbb{Z}_E$, and $m_{t_i} \in \mathbb{Z}_M$, $i = 1, 2$. The operation $\oplus$ is defined by*

$$t_1 \oplus t_2 = ((s_{t_1} + s_{t_2}) \bmod S, (e_{t_1} + e_{t_2}) \bmod E, (m_{t_1} + m_{t_2}) \bmod M).$$

Therefore, component-wise sharing of a floating point number f between two parties is defined by $f = (s_f, e_f, m_f) = t_1 \oplus t_2$, where t_1 is held by one party and t_2 is held by the other. Since each integer of the 3-tuple are additively shared, no information can be obtained one of them unless all shares are combined. Hence,

the component-wise shared floating point number f can only be recovered if and only if all shares are combined together.

Although additive sharing can also use a 3-tuple of integers in representing shared floating point numbers, they are incompatible with componentwise additive sharing, since the former combines the shared number by applying the normal floating point addition $+_{fp}$, and the latter combines them by the defined operation $\oplus$. Since the operation $\oplus$ is more simpler than normal floating point addition $+_{fp}$, which requires a series of operations such as alignment and normalization, componentwise additive sharing is less complex than additive sharing. Therefore, the input and output of our floating point protocols are all assumed to be in a componentwise additive sharing scheme.

4. The Floating Point Arithmetic Protocols

We introduce the floating point arithmetic protocols in this section. We also give the conversion method for additive sharing to componentwise additive sharing of a floating point number.

4.1. Preliminaries

The four basic arithmetic operations for floating point numbers are addition, subtraction, multiplication, and division. The inputs of each protocol are a pair of floating point numbers $f_i = (f_i.s, f_i.e, f_i.m)$, where $i = 1, 2$, $f_i.s$ is the sign bit, $f_i.e$ is the exponent, $f_i.m$ is the mantissa, and each is componentwise additive shared between the two parties. The output of each protocol is a floating point number, which is also componentwise additive shared.

Constituent of the Protocols			
Protocol	Domain	Constituent	Times
Product	$\mathbb{Z}_{M^2}$	Z_2-to-$Z_n(\ell_m)$	2
	$\mathbb{Z}_M$	Z_n-to-Z_2	2
	$\mathbb{Z}_{M^2}$	Product	1
	$\mathbb{Z}_{M^2}$	Z_n-to-Z_2	1
	$\mathbb{Z}_M$	Z_2-to-Z_n	2
	$\mathbb{Z}_M$	If-Then-Else	1
	$\mathbb{Z}_E$	If-Then-Else	1
Division	Z_2	Z_2-to-Z_n	2
	$\mathbb{Z}_{M^2}$	Z_n-to-$Z_2(2\ell_m - 2)$	1
	$\mathbb{Z}_{M^2}$	Z_n-to-$Z_2(\ell_m)$	1
	$\mathbb{Z}_{M^2}$	Comparison	$\ell_m - 1$
	$\mathbb{Z}_{M^2}$	If-Then-Else	$\ell_m - 1$
	$\mathbb{Z}_M$	Z_2-to-$Z_n(\ell_m - 2)$	2
	$\mathbb{Z}_M$	If-Then-Else	1
	$\mathbb{Z}_E$	If-Then-Else	1
Addition Subtraction	$\mathbb{Z}_E$	Z_n-to-Z_2	2
	$\mathbb{Z}_M$	Z_2-to-$Z_n(\ell_e)$	2
	$\mathbb{Z}_M$	Comparison	4
	$\mathbb{Z}_M$	Shift Right	3
	$\mathbb{Z}_M$	If-Then-Else	$\ell_m + 5$
	$\mathbb{Z}_M$	Z_n-to-Z_2	1
	$\mathbb{Z}_2$	Product	ℓ_m
	$\mathbb{Z}_E$	If-Then-Else	4
	$\mathbb{Z}_M$	Shift Left	1

Table 2: The constituents of the floating point protocol.

Complexity of the Protocols			
Protocol	Domain	Dimension	Times
Product	$\mathbb{Z}_{M^2}$	ℓ_m	2
	$\mathbb{Z}_2$	3	$4\ell_m - 3$
	$\mathbb{Z}_{M^2}$	2	1
	$\mathbb{Z}_M$	$\ell_m - 2$	2
	$\mathbb{Z}_M$	1	1
	$\mathbb{Z}_M$	2	1
	$\mathbb{Z}_E$	1	1
	$\mathbb{Z}_E$	2	1
Division	$\mathbb{Z}_2$	3	$2\ell_m^2 - \ell_m - 1$
	$\mathbb{Z}_2$	3	$2\ell_e + 15\ell_m - 17$
	$\mathbb{Z}_M$	ℓ_e	2
	$\mathbb{Z}_M$	1	$5\ell_m + 5$
	$\mathbb{Z}_M$	2	$13\ell_m + 9$
	$\mathbb{Z}_M$	ℓ_m	6
	$\mathbb{Z}_2$	2	ℓ_m
	$\mathbb{Z}_E$	1	4
	$\mathbb{Z}_E$	2	4
Addition Subtraction	$\mathbb{Z}_{M^2}$	ℓ_m	1
	$\mathbb{Z}_{M^2}$	$2\ell_m - 2$	1
	$\mathbb{Z}_{M^2}$	1	$2\ell_m - 1$
	$\mathbb{Z}_{M^2}$	2	$\ell_m - 1$
	$\mathbb{Z}_{M^2}$	$\ell_m - 2$	2
	$\mathbb{Z}_M$	1	1
	$\mathbb{Z}_M$	2	1
	$\mathbb{Z}_E$	1	1
	$\mathbb{Z}_E$	2	1

Table 3: The complexity of the floating point protocol. The complexity of the protocols are measured in the numbers of scalar product protocols executed.

The implementation of the protocols adopts the following convention and policy:

- Any kind of event that may trigger an exception, such as an overflow, are not handled, since we do not support any exception handling for reasons mentioned previously. The protocols continue to compute as normal but may produce erroneous results when such events occur.

- When a number x is converted to a bitwise-shared binary string str_x with length k by the Z_n-to-Z_2 protocol, the most significant bit is at $str_x[k-1]$, and the least significant bit is at $str_x[0]$.

The specification of the floating point arithmetic protocols are as follows:

Specification 2 (Floating Point Arithmetic Protocols)**.** *Party 1 and Party 2 componentwise additively shared two floating point numbers $f_1 = x_1 \oplus x_2$ and $f_2 = y_1 \oplus y_2$, where x_1 and y_1 are held by Party 1 , x_2 and y_2 are held by Party 2 . The operation $f_1 \circ f_2$ is performed by executing the protocol $((x_1, y_1), (x_2, y_2)) \mapsto (z_1, z_2)$, where z_1, z_2 are held by Party 1 and Party 2 , respectively, such that $z_1 \oplus z_2 = (x_1 \oplus x_2) \circ (y_1 \oplus y_2)$, and , for operator $\circ \in \{+_{fp}, -_{fp}, \times_{fp}, \div_{fp}\}$.*

We will now provide the implementation details in the following subsections. The constituent and complexity of the floating point arithmetic protocols are provided in Table 4.1.

4.2. The Addition and Subtraction Protocol

The implementation of the addition and subtraction protocol is essentially divided into four stages. From line 5 to line 9, an alignment of the mantissa is first performed. The addition or subtraction of the mantissa is then performed from line 12 to line 14. The sign bit, a preliminary mantissa, and a preliminary exponent are determined on line 17, 18 and 19. Since the preliminary mantissa and exponent may not be normalized, a postnormalization is performed from line 22 to line 38. The preliminary mantissa m_{origin} is first transformed to bitwise-shared binary string str_m to compute the value of the two flags $left_{shift}$ and $right_{shift}$, indicates whether a left or right shift operation for m_origin to be normalized. Subtraction is performed by simply inverting the sign bit of the desired subtrahend. The details of the addition protocol are described in Algorithm 1.

4.3. The Multiplication Protocol and the Division Protocol

The multiplication protocol first determines the sign bit and a preliminary exponent on line 5 and line 8, respectively. An expanding the mantissas to larger domain is performed on line 11 and line 12, and the resulting product of the two expanded mantissas is stored in m_{origin} on line 13. A postnormalization is performed from line 16 to line 19, and the msb used to determine if a shift operation is needed for the preliminary mantissa m_{origin} to be normalized. The details of the addition protocol are described in Algorithm 3.

The division protocol is very similar to that of the multiplication, consisting of determining sign and exponent value on line 5 and line 23, the expansion of the mantissas to larger domain from line 8 to 12, and postnormalization from line 26 to 28. The difference is that instead of performing multiplication, a division operation is performed on the mantissas from line 15 to line 21 by long division. Detailed description of the implementation of the division protocol is given in Algorithm 2.

4.4. Conversion of Data Sharing Formats

Since the input and output of our protocols are all in the componentwise additive sharing scheme and the initial data are assumed to be in the additive sharing scheme, a conversion that transforms additive sharing to componentwise additive sharing is needed for every private input before any computation can take place. The conversion is essentially to find a pair of 3-tuple integers s_1 and s_2, such that $f = f_1 +_{fp} f_2 = s_1 \oplus s_2$, where f_i and s_i are held by Party i.

The floating point number zero is $0 = (0, 0, 0)$ in both sharing formats. Hence for any floating point number f, the relation $f = f +_{fp} 0 = f \oplus 0$ holds. Therefore, by performing a single floating point addition protocol $add_{fp}((f_1, 0), (0, f_2)) \mapsto (s_1, s_2)$, which is effectively $f = f_1 +_{fp} f_2 = (f_1 +_{fp} 0) +_{fp} (0 +_{fp} f_2) = (f_1 \oplus 0) +_{fp} (0 \oplus f_2) = s_1 \oplus s_2$, the desired conversion is achieved.

Algorithm 1 Floating Point Addition Protocol

Input: $f_1 = (f_1.s, f_1.e, f_1.m)$, $f_2 = (f_2.s, f_2.e, f_2.m)$
Output: $f = (sign, exponent, mantissa)$

/* Align mantissa */
$e_1 \leftarrow Z_2\text{-to-}Z_n(Z_n\text{-to-}Z_2(f_1.e, \mathbb{Z}_E), \mathbb{Z}_M)$
$e_2 \leftarrow Z_2\text{-to-}Z_n(Z_n\text{-to-}Z_2(f_2.e, \mathbb{Z}_E), \mathbb{Z}_M)$
$d \leftarrow \text{Comparison}(e_1 - e_2)$
$m_1 \leftarrow \text{If-Then-Else}(d,\ \text{Shift-Right}(f_1.m, (e_2 - e_1)), f_1.m)$
$m_2 \leftarrow \text{If-Then-Else}(d,\ f_2.m,\ \text{Shift-Right}(f_2.m, (e_1 - e_2)))$

/* Add sign and perform addition on mantissa */
$m_1 \leftarrow \text{If-Then-Else}(f_1.s, -m_1, m_1)$
$m_2 \leftarrow \text{If-Then-Else}(f_2.s, -m_2, m_2)$
$m_{origin} \leftarrow m_1 + m_2$

/* Determine sign bit and decide a preliminary mantissa and exponent */
$sign \leftarrow \text{Comparison}(m_{orign})$
$m_{origin} \leftarrow \text{If-Then-Else}(sign, -m_{origin}, m_{origin})$
$e_{origin} \leftarrow \text{If-Then-Else}(d,\ f_2.e,\ f_1.e)$

/* Postnormalization */
$str_m[\ ] \leftarrow Z_n\text{-to-}Z_2(m_{origin}, \mathbb{Z}_M)$

$e_{lshift} \leftarrow e_{origin}$
$left_{shift} \leftarrow \text{Not}(\text{Or}(str_m[\ell_m - 1], str_m[\ell_m - 2]))$
$left_{shift} \leftarrow \text{And}(left_{shift}, \text{Not}(\text{Equal-Zero}(m_{origin})))$
$dis \leftarrow 0$, $lf \leftarrow \text{Not}(left_{shift})$
for $i \leftarrow (\ell_m - 2) \rightarrow 0$ **do**
  $lf \leftarrow \text{Addition}(lf, str_m[i])$ - $\text{Product}(lf, str_m[i])$
  $dis \leftarrow \text{If-Then-Else}(lf, dis, \text{Addition}(dis, \text{ONE}))$
end for
$e_{lshift} \leftarrow \text{If-Then-Else}(left_{shift}, (e_{lshift} - dis), e_{lshift})$
$m_{lshift} \leftarrow \text{Shift-Left}(m_{origin}, dis)$

$right_{shift} \leftarrow str_m[\ell_m - 1]$
$m_{rshift} \leftarrow \text{Shift-Right}(m_{origin}, \text{ONE})$
$m_{rshift} \leftarrow \text{If-Then-Else}(right_{shift}, m_{rshift}, m_{origin})$
$e_{rshift} \leftarrow \text{If-Then-Else}(right_{shift}, \text{Addition}(e_{origin}, \text{ONE}), e_{origin})$

$mantissa \leftarrow \text{If-Then-Else}(right_{shift}, m_{rshift}, m_{lshift})$
$exponent \leftarrow \text{If-Then-Else}(right_{shift}, e_{rshift}, e_{lshift})$

return $(sign, exponent, mantissa)$

Algorithm 2 Floating Point Division Protocol

Input: $f_1 = (f_1.s, f_1.e, f_1.m)$, $f_1 = (f_2.s, f_2.e, f_2.m)$
Output: $f = (sign, exponent, mantissa)$

/* Determine the sign bit */
$sign \leftarrow \text{Or}(f_1.s, f_2.s)$

/* Expand the domain of the mantissas */
$zero\_string[\ ] \leftarrow [0, 0 \cdots 0]$ /* the length is 24 */
$str_{m_1}[\ ] \leftarrow Z_n\text{-to-}Z_2(f_1.m, \mathbb{Z}_M)$
$str[\ ] \leftarrow \text{Concatinate\_array}(zero\_string[\ ], str_{m_1}[\ ])$
$m_1 \leftarrow Z_2\text{-to-}Z_n(str[\ ], \mathbb{Z}_{M^2})$
$m_2 \leftarrow Z_2\text{-to-}Z_n(Z_n\text{-to-}Z_2(f_2.m, \mathbb{Z}_M), \mathbb{Z}_{M^2})$

/* Perform Division */
$q\_array[\ ] \leftarrow [\ ]$ /* an empty array */
for $i \leftarrow (\ell_m - 2)$ **to** 0 **do**
  $t \leftarrow m_1 - (m_2 \cdot 2^i)$
  $bit \leftarrow \text{Comparison}(t)$
  $q\_array[i] \leftarrow \text{Addition}(bit, \text{ONE})$
  $m_1 \leftarrow \text{If-Then-Else}(bit, m_1, t)$
end for

$e_{origin} \leftarrow \text{Addition}(f_1.e - f_2.e, \text{BIAS})$

/* Postnormalization */
$msb \leftarrow q\_array[\ell_m - 2]$
$m_{shift} \leftarrow Z_2\text{-to-}Z_n(q\_array[(\ell_m - 1) \cdots 1], \mathbb{Z}_M)$
$m_{origin} \leftarrow Z_2\text{-to-}Z_n(q\_array[(\ell_m - 2) \cdots 0], \mathbb{Z}_M)$

$mantissa \leftarrow \text{If-Then-Else}(msb, m_{shift}, m_{origin})$
$exponent \leftarrow \text{If-Then-Else}(msb, e_{origin}, \text{Addition}(e_{origin}, -\text{ONE}))$

return $(sign, exponent, mantissa)$

Algorithm 3 Floating Point Multiplication Protocol

Input: $f_1 = (f_1.s, f_1.e, f_1.m)$, $f_1 = (f_2.s, f_2.e, f_2.m)$
Output: $f = (sign, exponent, mantissa)$

/*Determine the sign bit*/
$sign \leftarrow \text{Or}(f_1.s, f_2.s)$

/*Determine a preliminary exponent*/
$e_{origin} \leftarrow \text{Addition}((f_1.e + f_2.e), -\text{BIAS})$

/*Expand mantissas' domain and perform multiplication*/
$m_1 \leftarrow Z_2\text{-to-}Z_n(Z_n\text{-to-}Z_2(f_1.m, \mathbb{Z}_M), \mathbb{Z}_{M^2})$
$m_2 \leftarrow Z_2\text{-to-}Z_n(Z_n\text{-to-}Z_2(f_2.m, \mathbb{Z}_M), \mathbb{Z}_{M^2})$
$m_{origin} \leftarrow \text{Product}(m_1, m_2)$

/* Postnormalization */
$m\_array[\ ] \leftarrow Z_n\text{-to-}Z_2(m_{origin}, \mathbb{Z}_{M^2})$
$msb \leftarrow m\_array[2\ell_m - 1]$
$m_{shift} \leftarrow Z_2\text{-to-}Z_n(m\_array[(2\ell_m - 1) \cdots (\ell_m + 1)], \mathbb{Z}_M)$
$m_{origin} \leftarrow Z_2\text{-to-}Z_n(m\_array[(2\ell_m - 2) \cdots (\ell_m - 1)], \mathbb{Z}_M)$

$mantissa \leftarrow \text{If-Then-Else}(msb, m_{shift}, m_{origin})$
$exponent \leftarrow \text{If-Then-Else}(msb, \text{Addition}(e_{origin}, \text{ONE}), e_{origin})$

return $(sign, exponent, mantissa)$

5. Experimental Results

The commodity-based approach is adopted for the secure scalar product protocol implementation as mentioned in section 3.2. All of the protocols and related programs are implemented in Ruby (version 1.9.1 p378). The commodity server ran on a 2.93GHz FreeBSD machine. Party 1 ran on a 3.04GHz FreeBSD, and Party 2 ran on a 3.0GHz Linux machine. All machines have DDR2 800 memory.

5.1. Individual Protocol Performance

In the experiments, we compute the average execution time and the average relative error of each implemented protocol. The floating point operands are generated by a Ruby implemented random number generator within the range of 0 to $1,000,000$. The generated data are additively shared. The relative error is a comparison of the output value of the protocols and the resulting value of the same operands with the same operation in Ruby, which is also in double precision. The performance is evaluated by the execution time of each protocol for each client. Each reported performance result is the average execution time of 100 executions and they are shown in the Party 1 and Party 2 columns of Table 4. For comparison, we also conducted performance measurements with the same framework, but as all three parties are executed locally on a single machine, any external network influence is avoided. The machine is the machine that ran the "commodity server". The results are shown in the Party 1 OPC and Party 2 OPC columns of Table 4.

Protocol	Party 1	Party 2	Party 1 OPC.	Party 2 OPC.	Relative error
Add./Sub.	2.5441	2.5445	0.8006	0.8007	1.17965×10^{-16}
Product	0.9560	0.9564	0.090646	0.090433	2.05098×10^{-16}
Division	4.5889	4.5894	2.42991	2.429692	7.47230×10^{-17}

Table 4: Individual protocol performance measured in seconds. The second and third columns, the protocols are executed in a real network setting, while the fourth and fifth columns are executed locally on a single machine.

It can be seen that our execution time is practical and confirms our complexity analysis. The relative error may be caused by truncation, since the rounding mode of most floating point implementations are round-to-the-nearest, which is on average smaller than that of truncation[13], but it can be observed that the errors are reasonably small.

It is clear that the product protocol has the best performance. Although the product protocol has a similar complexity with the addition protocol in terms of the number of scalar products performed, as displayed in Table 3, the dimension and the domain of the vectors involved in the addition protocol are larger, and this will result in penalties on performance. The division protocol has the poorest performance, since the computation of the mantissa is costly.

5.2. Application of Floating Point Arithmetic Protocols

We now present a few applications of our floating point arithmetic protocols by performing various common statistic computation.

5.2.1. Computing Mean and Variance.

We first perform the computation of mean and variance. The computation is performed on a set of 100 floating point numbers, generated by a Ruby implemented random number generator within the range of 0 to $1,000,000$, and under componentwise additive sharing. The final results are compared to the value computed by one party computation, namely when a party performs the computation all by itself, which is also implemented in Ruby.

	Mean	Variance
Party 1	194.5968 sec.	194.5816 sec.
Party 2	610.5248 sec.	610.4716 sec.
Relative Error	4.60051×10^{-15}	3.77015×10^{-15}

Table 5: The results of the calculation of statistical values. The second column of the two regression experiments are the resulting coefficients of the Ruby implementation, while the third column are by the protocols.

Simple Linear Regression		
coeff.	Ruby OPC	MPC
α	29.917436789508	29.9174367890303
β	0.541747011438181	0.541747011445812
MSE	5.128696287	5.12869629
Multivariate Regression		
Heights and weights for American women		
coeff.	Ruby OPC	MPC
β_1	128.875848538853	128.875848538469
β_2	−143.245136869685	−143.245136869977
β_3	61.9876613958928	61.9876613954777
Randomly generated data set		
coeff.	Ruby OPC	MPC
β_1	0.387944654584357	0.387944654584364
β_2	0.355174454658603	0.355174454658616
β_3	0.350174271933872	0.350174271933895
β_4	0.348354902835034	0.348354902835044
β_5	0.344983551087036	0.344983551087051

Table 6: The results of the calculation of statistical values. The second column of the two regression experiments are the resulting coefficients of the Ruby implementation, while the third column are by the protocols.

The results are shown in Table 5. The table shows the execution time that is needed for both Party 1 and Party 2 to compute the mean and variance. It shows that our protocol performs reasonably well with low relative error.

5.2.2. Simple Linear Regression.

Next we give an implementation of a simple linear regression algorithm, which has many real-life applications [20]. We adopt the least square method for regression. Our regression model is $y = \alpha + \beta x$, and that $\hat{\alpha}$, $\hat{\beta}$ are determined by $\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}, \hat{\beta} = (n\sum x_i - \sum x_i y_i)/(n\sum x_i^2 - (\sum x_i)^2)$.

The data set for this experiment is 1373 sets of the heights of mothers in the UK under the age of 65 and one of their adult daughters over the age of 18 in the period of 1893-1898 collected by E.S. Pearson [20]. It is assumed that the computation is carried out under homogeneous sharing, and the distribution of the data sets among Party 1 and Party 2 is arbitrary since this setting is most commonly seen in practical settings. For this experiment Party 1 has 544 sets and Party 2 has 829 sets.

The results are shown in the first part of Table 6. The resulting parameters computed by the protocols are shown in the MPC column. For comparison, we also implement a simple linear regression program in Ruby with the same method and approach and applied it on to the same data set, and with the results in Ruby OPC column. Also, the MSE of each model are given on the third row. It can be observed that there are only minor differences with the model that is generated with the Ruby implementation, and it may also be caused by rounding errors.

5.2.3. Multivariate Linear Regression.

To further demonstrate the capability of our floating point protocols, we perform a more complex task of doing multivariate linear regression on two data sets [20]. We used the ordinary least squares (OLS) method, which is essentially the calculation of the coefficient matrix by $\hat{\beta} = (X^T X)^{-1} X^T y$.

Computations are carried out on two data sets. The data are homogeneous shared, with the distribution of the data sets among Party 1 and Party 2 being arbitrary. One of which is fitting 15 sets of heights and weights for American women aged 30 to 39 from *The World Almanac and Book of Facts (1975)* to the model $w_i = \beta_1 + \beta_2 h_i + \beta_3 h_i^2 + \epsilon_i$, where β_1, β_2, and β_3 are the coefficients. For this experiment Party 1 has 7 sets and Party 2 has 8 sets. The other is fitting 1000 sets of randomly generated numbers, generated by a Ruby implemented random generator, within the range of 0 to $1,000,000$ to the model $w_i = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4$, where $\beta_1, \beta_2, \beta_3$, and β_4 are the coefficients. Party 1 has 280 sets and Party 2 has 720 sets in this experiment.

The coefficients of the models are shown in Table 6. The resulting parameters computed by the protocols are shown in the MPC column. A program implemented in Ruby with the same method and approach is also applied to the same data set for comparison with the SMC results, and its results are shown in the Ruby OPC column.

It can be readily observed that apart from a small difference that may have been caused by truncation, there is no significant difference between the result of the two computations.

6. Conclusion

We have developed a set of floating point arithmetic protocols by utilizing secure scalar product protocols. The protocols can be used to construct secure data analysis applications across distributed databases, while preserving the privacy and integrity of the data. The design of the protocols allows the user to define their own floating point format, and strike their own balance between precision and performance.

The specific implementation of our protocols follows the double-precision floating point format in the IEEE-754 standard, which allows it to be utilized in a straight-forward fashion in the implementation of secure data analysis applications. The practicality of our protocols is demonstrated by performing the most common statistical calculations such as mean, variance, and linear regression. The results shows that the applications constructed by our protocols can achieve low relative errors, and displays promising potentials for constructing more complex applications for data mining and analysis.

We are currently constructing more frequently used protocols for data analysis, such as exponentiation and logarithms, based on our floating point protocols. Also we are in the process of enhancing the performance and extending the precision of our protocols. We believe that our floating point protocol suite will be a promising solution to striking the balance between data availability and privacy.

References

[1] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, August 2008.

[2] 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/`.

[3] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008.

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

[5] Yi-Ting Chiang, Da-Wei Wang, Churn-Jung Liau, and Tsan-Sheng Hsu. Secrecy of two-party secure computation. *Data and Applications Security XIX, Lecture Notes in Computer Science*, 3654:114–123, 2005.

[6] Wenliang Du and Zhijun Zhan. A practical approach to solve secure multi-party computation problems. In *NSPW '02: Proceedings of the 2002 Workshop on New Security Paradigms*, pages 127–135, New York, NY, USA, 2002. ACM Press.

[7] Pierre-Alain Fouque, Jacques Stern, and Geert-Jan Wackers. Cryptocomputing with rationals. In *Proceedings of the 6th international conference on Financial cryptography*, FC'02, pages 136–146, Berlin, Heidelberg, 2003. Springer-Verlag.

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

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

[10] Oded Goldreich. *Foundations of Cryptography, Volume II Basic Applications*. Cambridge University Press, 1st edition, 2004.

[11] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC '87: Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York, NY, USA, 1987. ACM Press.

[12] Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 451–462, 2010.

[13] Israel Koren. *Computer Arithmetic Algorithms*. AK Peters/CRC Press, 2nd edition edition, 2001.

[14] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In *USENIX Security '04: Proceedings of the 13th Symposium on Security, Usenix*, pages 287–302, 2004.

[15] Chih-Hao Shen, Justin Zhan, Tsan-Sheng Hsu, Churn-Jung Liau, and Da-Wei Wang. Scalar-product based secure two-party computation. In *GrC '08: IEEE International Conference on Granular Computing*, pages 556–561, Aug. 2008.

[16] Chih-Hao Shen, Justin Zhan, Da-Wei Wang, Tsan-Sheng Hsu, and Churn-JungLiau. Information-theoretically secure number-product protocol. In *ICMLC '07: International Conference on Machine Learning and Cybernetics*, volume 5, pages 3006–3011, 19-22 Aug. 2007.

[17] Da-Wei Wang, Churn-Jung Liau, Yi-Ting Chiang, and Tsan-Sheng Hsu. Information theoretical analysis of two-party secret computation. *Data and Applications Security XX, Lecture Notes in Computer Science*, 4127:310–317, 2006.

[18] I-Cheng Wang, Kung Chen, Tsan-Sheng Hsu, Churn-Jung Liau, Chih-Hao Shen, and Da-Wei Wang. Protocols for secure multi-party computation: Design, implementation and performance evaluation. *Technical Report*, 2010.

[19] I-Cheng Wang, Chih-Hao Shen, Tsan-Sheng Hsu, Churn-Chung Liau, Da-Wei Wang, and Justin Zhan. Towards empirical aspects of secure scalar product. In *ISA '08: IEEE International Conference on Information Security and Assurance*, pages 573–578, April 2008.

[20] Sanford Weisberg. *Applied Linear Regression*. Wiley, 3rd edition edition, 2005.

[21] Andrew C. Yao. Protocols for secure computation. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 160–164, November 1982.