THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A
Volume 21, Number 1/2020, pp. 21–28

OPTIMIZING SECURE FLOATING-POINT ARITHMETIC: SUMS, DOT PRODUCTS, AND POLYNOMIALS

Octavian CATRINA

Politehnica University of Bucharest, Faculty of Electronics, Telecommunications and Information Technology
Iuliu Maniu 1-3, 061071 Bucharest, Romania
E-mail: octavian.catrina@upb.ro

Abstract: Secure computation supports collaborative applications with private inputs and outputs, by providing cryptographic protocols that protect data privacy during the computation. Important applications, like statistical analysis, benchmarking, data mining, and optimizations, require accurate and efficient secure computation with floating-point numbers. Secure floating-point addition is the main performance bottleneck and severely affects the performance of tasks that involve many additions. We present in this paper optimized protocols for evaluating sums, dot products, and polynomials, that offer important performance gains with respect to generic constructions. These protocols are part of a framework for secure multiparty computation with real numbers, constructed using a small collection of building blocks based on Shamir secret sharing.

Key words: Secure multiparty computation, Secret sharing, Secure floating-point arithmetic, Multi-operand addition, Dot product, Polynomial evaluation.

1. INTRODUCTION

A privacy-preserving collaborative application allows a group of mutually distrustful parties to run a joint computation without having to reveal their private inputs. Secure computation supports such applications by providing cryptographic protocols that protect data privacy throughout the computation. Important applications, like statistical analysis, benchmarking, data mining, and optimizations, require computation with real numbers and need the accuracy and dynamic range offered by floating-point numbers [1, 3, 8, 15]. The deployment of these applications is hindered by the severe performance penalty caused by the cryptographic protocols. The main drawback of secure floating-point arithmetic is addition, which is much slower than the other operations [6], and thus severely degrades the performance of a broad range of important tasks. This motivates further research, to find solutions that better satisfy the performance requirements of the applications, using alternative data representations and tradeoffs between performance and accuracy [14].

In this paper, we focus on secure multi-operand floating-point addition and related tasks, like evaluating dot products and polynomials. We show protocols that compute these tasks much more efficiently than usual, generic constructions, alleviating to a large extent the bottleneck caused by floating-point addition. Moreover, this performance gain is achieved without loss of accuracy.

Comprehensive support for secure multiparty computation with real numbers is offered by two frameworks based on secret sharing (building blocks, arithmetic operations, and applications). The first framework relies on Shamir secret sharing [11] and related techniques [10]. Protocols for secure integer and fixed-point arithmetic were presented in [7, 9] and used in secure liner programming [8]. Follow-up work added secure floating-point arithmetic [2, 6] and other applications [1]. The building blocks offer variants optimized for round complexity (fewer interactions between parties) or communication complexity (less data exchanged by the parties) [7]. The

other framework, Sharemind, is based on additive secret sharing and offers protocols for secure computation with fixed-point and floating-point numbers, as well as alternative representations of real numbers [3, 15]. The protocols and the implementation are highly optimized for low communication complexity.

The protocols presented in this paper rely on the first framework, extended with building blocks and optimizations aimed at providing better support for secure computation with real numbers [5, 6]. The paper is structured as follows. Section 2 is an overview of the secure computation model, data encoding, and building blocks. The new protocols used for secure multi-operand floating-point addition are presented in Section 3, followed by protocols for evaluating dot products and polynomials, in Section 4, We summarize the main results in Section 5, including a preliminary analysis of protocol complexity and performance measurements.

2. PRELIMINARIES

The protocols are based on standard primitives for secure computation using Shamir secret sharing [11] and related techniques [10, 12]. They offer perfect or statistical privacy, assuming perfectly secure communication channels. Core primitives provide secure arithmetic in a finite field $\mathbb{F}$ using Shamir secret sharing over $\mathbb{F}$, with perfect privacy against a passive threshold adversary that corrupts t out of n parties: the parties follow strictly the protocol and any $t+1$ parties can reconstruct a secret, while t or less parties cannot distinguish it from random values in $\mathbb{F}$. The parties locally compute addition/subtraction of shared field elements by adding/subtracting their shares. Tasks that involve multiplication require interaction and are computed by dedicated protocols.

The limitations of secure arithmetic with shared data are overcome by combining secret sharing with additive or multiplicative hiding: for shared variable $[\![x]\!]$, the parties jointly generate a shared random value $[\![r]\!]$, compute $[\![y]\!] = [\![x]\!] + [\![r]\!]$ or $[\![y]\!] = [\![x]\!] \cdot [\![r]\!], x \neq 0$, and reveal y (similar to one-time pad encryption with key r).

The protocols are evaluated using two complexity metrics that focus on interaction. Communication complexity counts the invocations of 3 primitives during which every party sends a share to the others: input sharing, multiplication, and secret reconstruction. Round complexity is the number of sequential invocations. The first metric takes into account the amount of data and the local computation involved in the execution of the interactive primitives. The second one takes into account the effects of other communication delays. The pseudocode of the protocols is annotated with the number of rounds followed by the number of interactive primitives.

Interactive operations that do not depend on each other are executed in parallel, in a single round. In particular, all shared random values used by a secure computation task can be precomputed in parallel. Part of these shared random values are generated without interaction, using Pseudo-random Replicated Secret Sharing (PRSS) [10] and its integer variant (RISS) [13] (random field elements and integers, and random sharings of 0).

Boolean, integer, fixed-point, and floating-point data types are encoded as elements of a finite field $\mathbb{F}$. In this paper, all protocols use the field $\mathbb{Z}_q$, with prime q large enough for all data types. Different representations of a value are distinguished as follows: we denote $\tilde{x}$ a fixed-point number, $\bar{x}$ the integer value that encodes $\tilde{x}$, x the field element that encodes $\bar{x}$, and $[\![x]\!]$ a sharing of secret x; a floating-point number is denoted $\hat{x}$. The notation $x = (condition)?\ a : b$ means that x is assigned the value a when $condition = true$ and b otherwise.

Boolean values $false, true$ and bit values $0, 1$ are encoded as 0_F and 1_F, respectively. This encoding allows efficient secure evaluation of Boolean functions using secure arithmetic in $\mathbb{F}$ [7].

Signed integer types are defined as $\mathbb{Z}_{\langle k \rangle} = \{\bar{x} \in \mathbb{Z} \mid \bar{x} \in [-2^{k-1}, 2^{k-1} - 1]\}$. They are encoded in $\mathbb{Z}_q$ by the function $\mathsf{fld} : \mathbb{Z}_{\langle k \rangle} \mapsto \mathbb{Z}_q$, $\mathsf{fld}(\bar{x}) = \bar{x} \bmod q$, for $q > 2^{k+\kappa}$, where κ is the security parameter (negative integers $\bar{x} \in [-2^{k-1}, -1]$ are mapped to $x \in [q - 2^{k-1}, q - 1]$). This enables efficient secure integer arithmetic based on secure arithmetic in $\mathbb{Z}_q$: for any $\bar{x}_1, \bar{x}_2 \in \mathbb{Z}_{\langle k \rangle}$ and $\odot \in \{+, -, \cdot\}$, we compute $\bar{x}_1 \odot \bar{x}_2 = \mathsf{fld}^{-1}(\mathsf{fld}(\bar{x}_1) \odot \mathsf{fld}(\bar{x}_2))$; also, if $\bar{x}_2 \mid \bar{x}_1$ then $\bar{x}_1 / \bar{x}_2 = \mathsf{fld}^{-1}(\mathsf{fld}(\bar{x}_1) \cdot \mathsf{fld}(\bar{x}_2)^{-1})$.

Signed fixed-point types are rational numbers defined as $\mathbb{Q}^{FX}_{\langle k,f \rangle} = \{\tilde{x} \in \mathbb{Q} \mid \tilde{x} = \bar{x} 2^{-f}, \bar{x} \in \mathbb{Z}_{\langle k \rangle}\}$, for $f < k$. $\mathbb{Q}^{FX}_{\langle k,f \rangle}$ is mapped to $\mathbb{Z}_{\langle k \rangle}$ by the function $\mathsf{int} : \mathbb{Q}^{FX}_{\langle k,f \rangle} \mapsto \mathbb{Z}_{\langle k \rangle}$, $\bar{x} = \mathsf{int}_f(\tilde{x}) = \tilde{x} 2^f$, and then encoded in $\mathbb{Z}_q$ as described above. Secure fixed-point multiplication and division require $q > 2^{2k+\kappa}$.

Table 1
Complexity of the main building blocks used in this paper.

Protocol	Rounds	Inter. op.	Precomp.
Div2m($[\![a]\!],k,m$)	3	$m+2$	$3m$
Div2mP($[\![a]\!],k,m$)	1	1	m
Div2($[\![a]\!],k$)	1	1	1
LTZ($[\![a]\!],k$)	3	$k+1$	$3k$

Protocol	Rounds	Inter. op.	Precomp.
PreDiv2m($[\![a]\!],k,m$)	3	$2m+1$	$4m$
PreDiv2mP($[\![a]\!],k,m$)	1	1	m
SufOr($\{[\![a_i]\!]\}_{i=1}^{k}$)	2	$2k-1$	$3k$
Int2Mask($[\![x]\!],k$)	1	$k-1$	$2k-3$

Floating-point numbers $\hat{x} \in \mathbb{Q}^{FL}_{\langle l,g\rangle}$ are defined as tuples $\langle \bar{v},\bar{p},s,z\rangle$, where $\bar{v} \in [2^{\ell-1},2^{\ell}-1]\cup\{0\}$ is the unsigned, normalized significand, $\bar{p} \in \mathbb{Z}_{\langle g\rangle}$ is the signed exponent, $s = (\bar{v}<0)?\ 1:0$, and $z = (\bar{v}=0)?\ 1:0$, encoded in $\mathbb{Z}_q$ as described above. The value of the number is $\hat{x} = (1-2s)\cdot\bar{v}\cdot 2^{\bar{p}}$. If $\hat{x}=0$ then $\bar{v}=0$, $z=1$, and $\bar{p}=-2^{g-1}$. This encoding of $\hat{x}=0$ is justified by an efficiency tradeoff: it offers a significant gain for addition, the main performance bottleneck, with negligible loss for multiplication and division. Secure floating-point multiplication and division require $q > 2^{2\ell+\kappa}$.

Fixed-point and floating-point arithmetic protocols rely on a small set of building blocks introduced in [5,7], that efficiently compute $\bar{b} = \bar{a}\cdot 2^m$ and $\bar{c} \approx \bar{a}/2^m$, for secret $\bar{a},\bar{b},\bar{c} \in \mathbb{Z}_{\langle k\rangle}$ and public or secret integer $m \in [0,k-1]$. Table 1 lists their online complexity (rounds and interactive operations) and precomputation complexity.

If m is public, $\bar{a}\cdot 2^m$ is a local operation and $\bar{a}/2^m$ is computed by Div2m or Div2mP [7,9]. Div2m rounds to $-\infty$, while Div2mP rounds probabilistically to the nearest integer. We denote their outputs $\lfloor \bar{a}/2^m \rfloor$ and $\lfloor \bar{a}/2^m \rceil$, respectively. Div2mP computes $\bar{c} = \lfloor \bar{a}/2^m \rfloor + u$ and $u=1$ with probability $p = \frac{\bar{a} \bmod 2^m}{2^m}$ (e.g., if $\bar{a}=46$ and $m=3$ then $\bar{a}/2^m = 5.75$; the output is $\bar{c}=6$ with probability $p=0.75$ or $\bar{c}=5$ with probability $1-p=0.25$). For both variants, the rounding error is $|\delta|<1$ and $\delta=0$ if 2^m divides $\bar{a}$. Div2mP is much more efficient than Div2m and its output is likely more accurate. However, many applications need Div2m. For example, LTZ uses Div2m to compute $s = (\bar{a}<0)?\ 1:0 = -\lfloor \bar{a}/2^{k-1} \rfloor$.

If m is secret, we can use the following constructions. We start by computing the secret bits $\{x_i\}_{i=0}^{k-1}$, $x_i = (m=i)?\ 1:0$. Then, if we need $\bar{a}\cdot 2^m$, we compute $\bar{a}\cdot\sum_{i=0}^{k-1} x_i 2^i$. Otherwise, if we need $\bar{a}/2^m$, we compute the secret integers $\bar{d}_i = \{\lfloor \bar{a}/2^i \rfloor\}_{i=0}^{k-1}$ or $\bar{d}_i = \{\lfloor \bar{a}/2^i \rceil\}_{i=0}^{k-1}$ and $\sum_{i=0}^{k-1} x_i\bar{d}_i$. The efficiency of these constructions is improved by two other building blocks [5]. PreDiv2m, is a generalization of Div2m that computes $\{\lfloor \bar{a}/2^i \rfloor\}_{i=1}^{m}$. Surprisingly, it needs the same number of rounds and only m additional interactive operations. Similarly, PreDiv2mP is a generalization of Div2mP that computes $\{\lfloor \bar{a}/2^i \rceil\}_{i=1}^{m}$ with the same complexity as Div2mP.

Given a secret integer $\bar{x} \in [0,k-1]$, Protocol 1, Int2Mask, computes the secret bits $z_i = (\bar{x}=i)?\ 1:0$, $i \in [0,k-1]$, by Lagrange polynomial interpolation. Let $\alpha = x+1 \in [1,k]$. We can compute $\{z_i\}_{i=0}^{k-1}$ by evaluating the functions $f_i : [1,k] \to \{0,1\}$, $f_i(\alpha) = (\alpha = i+1)?\ 1:0$, for $i \in [0,k-1]$, using the polynomials $f_i(\alpha) = \sum_{j=0}^{k-1} a_{i,j}\alpha^j$. The coefficients $a_{i,j}$ are pre-computed from public information. Int2Mask computes $\{\alpha^i\}_{i=1}^{k-1}$ using PreMul [7] and then $z_i = f_i(\alpha)$, for $i \in [0,k-1]$ (PreMul requires non-zero inputs).

Protocol 1: $\{[\![z_i]\!]\}_{i=0}^{k-1} \leftarrow$ Int2Mask($[\![x]\!],k$)

1 $\{[\![y_j]\!]\}_{j=1}^{k-1} \leftarrow$ PreMul($\{[\![x]\!]+1\}_{i=1}^{k-1}$); // $1;k-1$
2 foreach $i \in [0,k-1]$ do $[\![z_i]\!] \leftarrow a_{i,0} + \sum_{j=1}^{k-1} a_{i,j}[\![y_j]\!]$;
3 return $\{[\![z_i]\!]\}_{i=0}^{k-1}$;

With standard protocols, multiplication followed by additive hiding, $d \leftarrow$ Reveal($[\![a]\!][\![b]\!]+[\![r]\!]$), needs 2 interactions and 2 rounds. We can avoid the first interaction by randomizing the share products: for all $i \in [1,n]$, party i computes $[\![c]\!]_{2t,i} \leftarrow [\![a]\!]_i[\![b]\!]_i + [\![0]\!]_{2t,i}$, where $[\![0]\!]_{2t,i}$ are pseudo-random shares of 0 generated with PRZS($2t$) [10]. We denote $[\![a]\!] * [\![b]\!]$ this local operation. Now the computation needs a single interaction: $d \leftarrow$ RevealD($[\![a]\!] * [\![b]\!] + [\![r]\!]$), where RevealD is the secret reconstruction protocol for polynomials of degree $2t$. Most of the protocols listed in Table 1 start with additive hiding of the input. We add variants of these protocols for input shared with a random polynomial of degree $2t$, and distinguish them by the suffix 'D'.

3. SECURE MULTI-OPERAND ADDITION

Our main goal is to efficiently compute multi-operand addition, $\hat{a} = \sum_{i=0}^{m-1} \hat{a}_i$, for secret $\{\hat{a}_i\}_{i=0}^{m-1} \in \mathbb{Q}_{\langle \ell,g \rangle}^{FL}$, $\hat{a}_i = (1-2s_i)\bar{v}_i 2^{\bar{p}_i}$ and secret $\hat{a} \in \mathbb{Q}_{\langle \ell,g \rangle}^{FL}$, $\hat{a} = (1-2s)\bar{v}2^{\bar{p}}$. The simplest approach is to use the protocol AddFL [6] and a generic algorithm for $m > 2$ operands with binary tree structure. The algorithm computes $m-1$ additions, which is optimal, in $\lceil \log m \rceil$ iterations. An iteration takes as input a vector, splits it into pairs of elements, adds them in parallel, and returns the results as a vector of half length. This is much better than sequential summation, which computes $m-1$ additions in $m-1$ iterations. Moreover, pairwise addition improves the accuracy, because each input is involved in $\log m$ additions instead of $m-1$ additions. The computation is shown in Protocol 2, SumGFL. Using a slightly optimized variant of AddFL, with the complexity shown in Table 2, the complexity of SumGFL is $16\lceil \log m \rceil$ rounds and $(m-1)(6\ell+3g+26)$ interactive operations.

Protocol 2: $\mathsf{SumGFL}(\{[\![v_i]\!],[\![p_i]\!],[\![s_i]\!]\}_{i=0}^{m-1},\ell,g)$

$\theta = \lceil \log m \rceil$; $\alpha = m$; $u = 0$;
foreach $i \in [1,\theta]$ do
  $\beta \leftarrow \lfloor \alpha/2 \rfloor$; $u \leftarrow \alpha \bmod 2$;
  foreach $j \in [1,\beta]$ do $([\![v_j]\!],[\![p_j]\!],[\![s_j]\!],[\![z_j]\!]) \leftarrow \mathsf{AddFL}(\{[\![v_k]\!],[\![p_k]\!],[\![s_k]\!]\}_{k=2j-1}^{2j},\ell,g)$; // Table 2
  if $u = 1$ then $([\![v_{\beta+1}]\!],[\![p_{\beta+1}]\!],[\![s_{\beta+1}]\!]) \leftarrow ([\![v_\alpha]\!],[\![p_\alpha]\!],[\![s_\alpha]\!])$;
  $\alpha \leftarrow \beta + u$;
return $([\![v_1]\!],[\![p_1]\!],[\![s_1]\!],[\![z_1]\!])$;

If $\bar{p}_1 \geq \bar{p}_2$, AddFL aligns the radix point and adds the significands by computing $\bar{p}'_3 = \bar{p}_1$ and $\bar{v}'_3 = (1-2s_1)\bar{v}_1 + \lfloor (1-2s_2)\bar{v}_2/2^{|\bar{p}_1-\bar{p}_2|} \rceil$. Then, it normalizes $\bar{v}'_3$ and $\bar{p}'_3$ to obtain $\hat{a} \in \mathbb{Q}_{\langle \ell,g \rangle}^{FL}$, encoded as in Section 2. If $\bar{p}_1 < \bar{p}_2$ the inputs are obliviously swapped. The normalization is expensive, but multi-operand addition can normalize only the final result, as follows. We want $\bar{v}$, $\bar{p}$, and s so that $\hat{a} = (1-2s)\bar{v}2^{\bar{p}} = \sum_{i=0}^{m-1}(1-2s_i)\bar{v}_i 2^{\bar{p}_i}$. Let $I = [0,m-1]$ and $x = \mathrm{argmax}_{i\in I}(p_i)$. Protocol 3, SumFL computes $\hat{a} = 2^{\bar{p}_x}\sum_{i\in I}(1-2s_i)\bar{v}_i/2^{\bar{p}_x-\bar{p}_i}$, as follows: step 1 finds $\bar{p}' = \bar{p}_x = \max(\{p_i\}_{i=0}^{m-1})$ using Protocol 4, MaxInt; steps 2-9 compute $\bar{v}' = \sum_{i=0}^{m-1} \lfloor \bar{v}'_i/2^{\bar{p}'-\bar{p}_i} \rceil$ by generalizing the algorithm used in AddFL; step 10 normalizes the output using the protocol FX2FL [6].

Protocol 3: $\mathsf{SumFL}(\{[\![v_i]\!],[\![p_i]\!],[\![s_i]\!]\}_{i=0}^{m-1},\ell,g)$

1 $[\![p']\!] \leftarrow \mathsf{MaxInt}(\{[\![p_i]\!]\}_{i=0}^{m-1},g)$; // $4\lceil \log m \rceil;(m-1)(g+2)$
2 foreach $i \in [0,m-1]$ do
3 $[\![c_i]\!] \leftarrow \mathsf{LTZ}([\![p']\!]-[\![p_i]\!]-\ell-1,g+1)$; // $3;m(g+2)$
4 $[\![\Delta'_i]\!] \leftarrow [\![p']\!]-[\![p_i]\!]$;
5 $[\![v'_i]\!] \leftarrow [\![v_i]\!](1-2[\![s_i]\!])$; // $0;m$
6 foreach $i \in [0,m-1]$ do
7 $\{[\![y_{i,j}]\!]\}_{j=0}^{\ell} \leftarrow \mathsf{Int2Mask}([\![\Delta'_i]\!],\ell+1)$; // $1;m\ell$
8 $[\![d_{i,0}]\!] \leftarrow [\![v'_i]\!]$; $\{[\![d_{i,j}]\!]\}_{j=1}^{\ell} \leftarrow \mathsf{PreDiv2mPD}([\![v'_i]\!]*[\![c_i]\!],\ell+1,\ell)$; // $0;m$
9 foreach $i \in [0,m-1]$ do $[\![v'_i]\!] \leftarrow \sum_{j=0}^{\ell}[\![y_{i,j}]\!][\![d_{i,j}]\!]$; // $1;m-1$
10 $([\![v]\!],[\![p]\!],[\![s]\!],[\![z]\!]) \leftarrow \mathsf{FX2FL}(\sum_{i=0}^{m-1}[\![v'_i]\!],[\![p']\!],\ell+\lceil \log m \rceil+1,0,\ell)$; // $9;5(\ell+\log m)$
11 return $([\![v]\!],[\![p]\!],[\![s]\!],[\![z]\!])$;

We could eliminate the iteration $i = x$ in steps 2-9 by extracting $\bar{v}_x$ and $\bar{p}_x$ from the input vector and computing $\hat{a} = 2^{\bar{p}_x}((1-2s_x)\bar{v}_x + \sum_{i\in I'}(1-2s_i)\bar{v}_i/2^{\bar{p}_x-\bar{p}_i})$, $I' = I \setminus \{x\}$. The iteration costs $\ell+g+3$ interactive operations, while extracting $\bar{v}_x$ and $\bar{p}_x$ takes $\lceil \log m \rceil + 2$ rounds and $3(m-2)$ operations. The simpler solution used in SumFL offers a better tradeoff for large m. However, AddFL and SumGFL avoid this redundant computation and are more efficient for $m = 2$. Accuracy in special cases can be improved as in AddFL [6].

The downside of normalizing only the final result is that steps 9-10 compute with larger integers: $\bar{v}'$ is a signed integer and $|\bar{v}'| \leq m \cdot 2^{\ell}$, so its maximum bit-length is $k = \ell + \lceil \log m \rceil + 1$. This increases the complexity

of the normalization. The complexity of SumFL is $4\lceil\log m\rceil+15$ rounds and about $(m+5)\ell+2mg+7m$ interactive operations, much better than SumGFL for $m>2$.

The protocol MaxInt computes $\bar{a}=\max(\{\bar{a}_i\}_{i=1}^m)$ with secret inputs (k-bit integers) and secret output using a generic binary tree construction. Each iteration takes as input a vector, splits it into pairs of elements, selects in parallel the larger value in each pair, and returns the results as a vector of half length ($\mathsf{GTZ}([\![x]\!],k)=\mathsf{LTZ}(-[\![x]\!],k)$). MaxInt computes $m-1$ comparisons and field multiplications in $\lceil\log m\rceil$ iterations, so its complexity is $4\lceil\log m\rceil$ rounds and $(m-1)(k+3)$ interactive operations.

Protocol 4: $[\![a]\!]\leftarrow\mathsf{MaxInt}(\{[\![a_i]\!]\}_{i=1}^m,k)$

1 $\theta=\lceil\log m\rceil$; $\alpha=m$; $u=0$;
2 foreach $i\in[1,\theta]$ do
3 $\quad\beta\leftarrow\lfloor\alpha/2\rfloor$; $u\leftarrow\alpha \bmod 2$;
4 $\quad$foreach $j\in[1,\beta]$ do $[\![b_{i,j}]\!]\leftarrow\mathsf{GTZ}([\![a_{2j-1}]\!]-[\![a_{2j}]\!],k+1)$; // $3\lceil\log m\rceil;(m-1)(k+2)$
5 $\quad$foreach $j\in[1,\beta]$ do $[\![a_j]\!]\leftarrow[\![b_{i,j}]\!]([\![a_{2j-1}]\!]-[\![a_{2j}]\!])+[\![a_{2j}]\!]$; // $\lceil\log m\rceil;m-1$
6 $\quad$if $u=1$ then $[\![a_{\beta+1}]\!]\leftarrow[\![a_\alpha]\!]$;
7 $\quad\alpha\leftarrow\beta+u$;
8 return $[\![a]\!]\leftarrow[\![a_1]\!]$;

4. APPLICATIONS

A first application is the evaluation of the dot product of two vectors, $\hat{a}=\sum_{i=0}^{m-1}\hat{a}_i\hat{x}_i$, for secret $\{\hat{a}_i\}_{i=0}^{m-1}\in\mathbb{Q}_{\langle\ell,g\rangle}^{FL}$, $\hat{a}_i=(1-2s_i)\bar{v}_i2^{\bar{p}_i}$, $\{\hat{x}_i\}_{i=0}^{m-1}\in\mathbb{Q}_{\langle\ell,g\rangle}^{FL}$, $\hat{x}_i=(1-2s_i')\bar{v}_i'2^{\bar{p}_i'}$, and secret $\hat{a}\in\mathbb{Q}_{\langle\ell,g\rangle}^{FL}$, $\hat{a}=(1-2s)\bar{v}2^{\bar{p}}$.

Floating-point multiplication is efficiently computed by the protocol MulFL [6]. The protocol computes $\bar{v}_3=\bar{v}_1\bar{v}_2$, $\bar{p}_3=\bar{p}_1+\bar{p}_2$, $s=s_1\oplus s_2$, $z=z_1\vee z_2$ and normalizes the result. Since $\bar{v}_1,\bar{v}_2\in[2^{\ell-1},2^\ell-1]\cup\{0\}$ and $\bar{v}_3\in[2^{2\ell-2},2^{2\ell}-2^{\ell+1}+1]\cup\{0\}$, normalization is simpler than for addition: if $\bar{v}_3<2^{2\ell-1}$ then $\bar{v}=\lfloor\bar{v}_3/2^{\ell-1}\rfloor$ and $\bar{p}=\bar{p}_3+\ell-1$, otherwise $\bar{v}=\lfloor\bar{v}_3/2^\ell\rfloor$ and $\bar{p}=\bar{p}_3+\ell$.

We can evaluate $\hat{a}=\sum_{i=0}^{m-1}\hat{a}_i\hat{x}_i$ using a generic protocol, DotPGFL, that computes $\{\hat{y}_i\}_{i=0}^{m-1}=\{\hat{a}_i\hat{x}_i\}_{i=0}^{m-1}$ in parallel using MulFL and then $\hat{a}=\sum_{i=0}^{m-1}\hat{y}_i$ using SumGFL. Its complexity, dominated by SumGFL, is $16\lceil\log m\rceil+5$ rounds and about $(m-1)(7\ell+3g+33)$ interactive operations. However, a more efficient solution combines MulFL and SumFL as shown in Protocol 5, DotPFL.

Protocol 5: $\mathsf{DotPFL}(\{[\![v_i]\!],[\![p_i]\!],[\![s_i]\!],[\![z_i]\!],[\![v_i']\!],[\![p_i']\!],[\![s_i']\!],[\![z_i']\!]\}_{i=0}^{m-1},\ell,g)$

1 foreach $i\in[0,m-1]$ do
2 $\quad[\![v_i'']\!]\leftarrow\mathsf{Div2mPD}([\![v_i]\!]*[\![v_i']\!],2\ell,\ell-1)$; // $1;m$
3 $\quad[\![s_i'']\!]\leftarrow[\![s_i]\!]\oplus[\![s_i']\!]$; $[\![z_i'']\!]\leftarrow[\![z_i]\!]\vee[\![z_i']\!]$; // $0;2m$
4 $\quad[\![p_i'']\!]\leftarrow([\![p_i]\!]+[\![p_i']\!]+\ell-1)(1-[\![z_i'']\!])-[\![z_i'']\!]2^{g-1}$; // $1;m$
5 $([\![v]\!],[\![p]\!],[\![s]\!],[\![z]\!])\leftarrow\mathsf{SumFL}(\{[\![v_i'']\!],[\![p_i'']\!],[\![s_i'']\!]\}_{i=0}^{m-1},\ell+1,\ell,g)$; // Table 2
6 return $([\![v]\!],[\![p]\!],[\![s]\!],[\![z]\!])$;

DotPFL computes $\langle\bar{v},\bar{p},s,z\rangle$ so that $(1-2s)\bar{v}2^{\bar{p}}=\sum_{i=0}^{m-1}(1-2s_i)(1-2s_i')\bar{v}_i\bar{v}_i'2^{\bar{p}_i+\bar{p}_i'}$ and $z=(\bar{v}=0)?\ 1:0$. Steps 1-4 compute in parallel $\bar{v}_i''=\lfloor\bar{v}_i\bar{v}_i'/2^{\ell-1}\rceil$, $s_i''=s_i\oplus s_i'$, and $z_i''=z_i\vee z_i'$, then $\bar{p}_i''=(z=0)?\ (\bar{p}_i+\bar{p}_i'+\ell-1):(-2^{g-1})$. DotPFL skips the relatively expensive normalization in MulFL. Instead, step 5 uses a slightly modified variant of SumFL that computes $\sum_{i=0}^{m-1}(1-2s_i'')\bar{v}_i''2^{\bar{p}_i''}$ for ℓ'-bit significands and then normalizes the output to ℓ bits. Here, $\ell'=\ell+1$, because $\bar{v}_i''\in[2^{\ell-1},2^{\ell+1}-2^2]\cup\{0\}$.

The online complexity of DotPFL is almost the same as for SumFL. Overall, it needs $4\lceil\log m\rceil+17$ rounds and about $(m+5)\ell+2mg+12m$ interactive operations, much less than DotPGFL.

Our next task is to evaluate a polynomial of degree m, $\hat{y}=P(\hat{x})=\sum_{i=0}^m\hat{a}_i\hat{x}^i$ with secret coefficients, variable, and output: $\{\hat{a}_i\}_{i=0}^m\in\mathbb{Q}_{\langle\ell,g\rangle}^{FL}$, $\hat{a}_i=(1-2s_i)\bar{v}_i2^{\bar{p}_i}$, $\hat{x}\in\mathbb{Q}_{\langle\ell,g\rangle}^{FL}$, $\hat{x}=(1-2s)\bar{v}2^{\bar{p}}$, and $\hat{y}\in\mathbb{Q}_{\langle\ell,g\rangle}^{FL}$, $\hat{y}=(1-2s')\bar{v}'2^{\bar{p}'}$.

Traditional algorithms are often optimized for assumptions that do not apply to secure computation. Horner's polynomial evaluation is sequential [16], so it is not suitable for secure computation. Also, Estrin's parallel polynomial evaluation [16] offers only a modest improvement for secure floating-point arithmetic. A better option is to first compute $\{\hat{x}_i\}_{i=1}^m = \{\hat{x}^i\}_{i=1}^m$ and then $\hat{a} = \hat{a}_0 + \sum_{i=1}^m \hat{a}_i\hat{x}_i$, using DotPGFL or DotPFL.

Protocol 6, PowAllFL, computes $\{\hat{x}^i\}_{i=1}^m$ with secret inputs and outputs. The protocol computes $m-1$ floating-point multiplications, which is optimal, in $\alpha = \lceil \log m \rceil$ iterations. Iteration $i \in [1, \alpha-1]$ computes in parallel $\gamma = 2^{i-1}$ multiplications, $\hat{x}_{\gamma+j} = \hat{x}_\gamma \hat{x}_j = \hat{x}^{\gamma+j}$, for $j \in [1,\gamma]$. Iteration α computes the remaining $m - 2^{\alpha-1}$ multiplications. For $\hat{x}_i = \hat{x}^i$, $s_i = (i \bmod 2 = 1)?\ s : 1$ and $z_i = z$, so PowAllFL uses a simplified variant of MulFL, called MulFLS, that does not compute s_i and z_i. The complexity of PowAllFL is $5\lceil \log m \rceil$ rounds and $(m-1)(\ell+5)$ interactive operations.

Protocol 6: PowAllFL($[\![v]\!], [\![p]\!], [\![s]\!], [\![z]\!], m, \ell, g$)

1 $([\![v_1]\!], [\![p_1]\!]) \leftarrow ([\![v]\!], [\![p]\!])$; $\alpha \leftarrow \lceil \log m \rceil$;
2 foreach $i \in [1, \alpha]$ do
3 $\gamma = 2^{i-1}$; $\beta \leftarrow (i < \alpha)?\ \gamma : m - \gamma$;
4 foreach $j \in [1, \beta]$ do
5 $([\![v_{\gamma+j}]\!], [\![p_{\gamma+j}]\!]) \leftarrow \mathsf{MulFLS}([\![v_\gamma]\!], [\![p_\gamma]\!], [\![v_j]\!], [\![p_j]\!], [\![z]\!], \ell, g)$; // $5\lceil \log m \rceil; (m-1)(\ell+5)$
6 $[\![s_{\gamma+j}]\!] \leftarrow ((\gamma + j) \bmod 2 = 1)?\ [\![s]\!] : 0$;
7 return $\{[\![v_i]\!], [\![p_i]\!], [\![s_i]\!], [\![z]\!]\}_{i=1}^m$;

We call PolySGFL a protocol that evaluates $\sum_{i=0}^m \hat{a}_i\hat{x}^i$ using PowAllFL and DotPGFL. PolySGFL computes $2m-1$ multiplications and m additions of floating-point numbers in $21\lceil \log m \rceil + 5$ rounds, with $m(8\ell + 3g + 41)$ interactive operations. We can improve PolySGFL by avoiding the normalization of intermediate results. Protocol 7, PolySFL, combines PowAllFL and DotPFL and simplifies the computation of $\hat{y}_i = \hat{a}_i\hat{x}^i = \langle \bar{v}_i'', \bar{p}_i'', s_i'', z_i'' \rangle$ by merging the steps that compute $\bar{p}_i''$, s_i'', and z_i''. PowAllFLS is a variant of PowAllFL that skips the operations that become redundant. The complexity of PolySFL is $9\lceil \log m \rceil + 16$ rounds and $(2m+4)\ell + 2mg + 16m$ interactive operations, much less than PolySGFL.

Protocol 7: PolySFL($[\![v]\!], [\![p]\!], [\![s]\!], [\![z]\!], \{[\![v_i]\!], [\![p_i]\!], [\![s_i]\!], [\![z_i]\!]\}_{i=0}^m, \ell, g$)

1 $(\{[\![v_i']\!], [\![p_i']\!]\}_{i=1}^m) \leftarrow \mathsf{PowAllFLS}([\![v]\!], [\![p]\!], m, \ell, g)$; // $5\lceil \log m \rceil; (m-1)(\ell+4)$
2 foreach $i \in [1, m]$ do $[\![z_i'']\!] \leftarrow [\![z]\!] \vee [\![z_i]\!]$; // $0; m$
3 foreach $i \in [1, m]$ do
4 $[\![v_i'']\!] \leftarrow \mathsf{Div2mPD}([\![v_i]\!] * [\![v_i']\!], 2\ell, \ell - 1)$; // $1; m$
5 $[\![p_i'']\!] \leftarrow ([\![p_i]\!] + [\![p_i']\!] + \ell - 1)(1 - [\![z_i'']\!]) - [\![z_i'']\!] 2^{g-1}$; // $0; m$
6 $[\![s_i'']\!] \leftarrow (i \bmod 2 = 1)?\ [\![s_i]\!] : [\![s]\!] \oplus [\![s_i]\!]$; // $0; m/2$
7 $([\![v_0'']\!], [\![p_0'']\!], [\![s_0'']\!]) \leftarrow ([\![v_0]\!], [\![p_0]\!], [\![s_0]\!])$;
8 $([\![v']\!], [\![p']\!], [\![s']\!], [\![z']\!]) \leftarrow \mathsf{SumFL}(\{[\![v_i'']\!], [\![p_i'']\!], [\![s_i'']\!]\}_{i=0}^m, \ell + 1, \ell, g)$; // Table 2
9 return $([\![v']\!], [\![p']\!], [\![s']\!], [\![z']\!])$;

Protocol 8: PowAllFLS($[\![v]\!], [\![p]\!], m, \ell, g$)

1 $([\![v_1]\!], [\![p_1]\!]) \leftarrow ([\![v]\!], [\![p]\!])$; $\alpha \leftarrow \lceil \log m \rceil$;
2 foreach $i \in [1, \alpha]$ do
3 $\gamma = 2^{i-1}$; $\beta \leftarrow (i < \alpha)?\ \gamma : m - \gamma$;
4 foreach $j \in [1, \beta]$ do
5 $([\![v_j']\!], [\![v_j'']\!]) \leftarrow \mathsf{SelDiv2mPD}([\![v_\gamma]\!] * [\![v_j]\!], 2\ell, \{\ell - 1, \ell\})$; // $\lceil \log m \rceil; m - 1$
6 $[\![b_j]\!] \leftarrow \mathsf{LTZ}([\![v_j']\!] - 2^\ell, \ell + 1)$; // $3\lceil \log m \rceil; (m-1)(\ell+2)$
7 $[\![v_{\gamma+j}]\!] \leftarrow [\![b_j]\!]([\![v_j']\!] - [\![v_j'']\!]) + [\![v_j'']\!]$; // $\lceil \log m \rceil; m - 1$
8 $[\![p_{\gamma+j}]\!] \leftarrow [\![p_\gamma]\!] + [\![p_j]\!] + \ell - [\![b_j]\!]$;
9 return $\{[\![v_i]\!], [\![p_i]\!]\}_{i=1}^m$;

Table 2

Complexity of the floating-point protocols.

Protocol	Task	Rounds	Interactive operations	Precomputation	Modulus
AddFL	$\hat{a} \leftarrow \hat{a}_1 + \hat{a}_2$	16	$6\ell + 3g + 29$	$\approx 13\ell + 9g$	$q > 2^{\ell+\kappa+1}$
MulFL	$\hat{a} \leftarrow \hat{a}_1 \hat{a}_2$	5	$\ell + 7$	$4\ell + 3$	$q > 2^{2\ell+\kappa}$

Protocol	Task	Rounds	Interactive operations	Modulus
SumGFL	$\hat{a} \leftarrow \sum_{i=0}^{m-1} \hat{a}_i$	$16\lceil \log m \rceil$	$6(m-1)\ell + (m-1)(3g+26)$	$q > 2^{\ell+\kappa+1}$
SumFL	$\hat{a} \leftarrow \sum_{i=0}^{m-1} \hat{a}_i$	$4\lceil \log m \rceil + 15$	$(m+5)\ell + m(2g+7)$	$q > 2^{\ell+\kappa+\log m+1}$
DotPGFL	$\hat{c} \leftarrow \sum_{i=0}^{m-1} \hat{a}_i \hat{b}_i$	$16\lceil \log m \rceil + 5$	$7(m-1)\ell + (m-1)(3g+33)$	$q > 2^{2\ell+\kappa}$
DotPFL	$\hat{c} \leftarrow \sum_{i=0}^{m-1} \hat{a}_i \hat{b}_i$	$4\lceil \log m \rceil + 17$	$(m+5)\ell + m(2g+11)$	$q > 2^{2\ell+\kappa}$
PolySGFL	$\hat{y} \leftarrow \sum_{i=0}^{m} \hat{a}_i \hat{x}^i$	$21\lceil \log m \rceil + 5$	$m(8\ell + 3g + 41)$	$q > 2^{2\ell+\kappa}$
PolySFL	$\hat{y} \leftarrow \sum_{i=0}^{m} \hat{a}_i \hat{x}^i$	$9\lceil \log m \rceil + 16$	$(2m+4)\ell + m(2g+15)$	$q > 2^{2\ell+\kappa}$

Table 3

Running time of the floating-point protocols (milliseconds).

	$m = 4$	Prec.	$m = 8$	Prec.	$m = 16$	Prec.	$m = 32$	Prec.
SumGFL	7.65	11.11	17.20	25.92	34.49	56.26	66.57	116.52
SumFL	5.47	7.02	9.06	11.63	15.86	21.08	28.21	39.10
DotPGFL	8.88	15.03	18.91	33.79	35.98	71.66	70.97	131.60
DotPFL	6.16	8.22	9.54	14.61	16.67	25.22	30.26	48.29
PolySGFL	12.51	21.41	23.98	43.96	43.25	89.72	81.31	157.76
PolySFL	8.26	12.75	12.71	21.98	21.52	41.29	37.52	80.93

3. EXPERIMENTAL RESULTS AND CONCLUSIONS

Floating-point addition is the main performance bottleneck in secure arithmetic, due to expensive operations for aligning the radix point and normalizing the result. We showed in this paper that the efficiency of secure multi-operand floating-point addition and related tasks, like dot product and polynomial evaluation, can be improved far beyond the efficiency of usual, generic constructions, without reducing the accuracy.

The protocols focus on improving the performance, accuracy and flexibility of secure computation tasks, rather than strictly following the IEEE 754 Standard for Floating-Point Arithmetic. The configuration parameters ℓ and g determine the precision and range of the numbers, as well as the protocol complexity (field size and number of interactive operations and rounds). The protocols work for any practically relevant values of these parameters, including IEEE 754 simple and double precision. The applications can choose ℓ and g according to their requirements, to obtain the best tradeoff between accuracy and performance.

Table 2 summarizes the online complexity of the floating-point protocols. SumGFL, the generic construction for computing $\sum_{i=0}^{m-1} \hat{a}_i$, reduces the round complexity with respect to sequential addition from $16(m-1)$ to $16\lceil \log m \rceil$ rounds. The optimized variant, SumFL needs only $4\lceil \log m \rceil + 15$ rounds and also substantially reduces the communication complexity. DotPFL, the optimized protocol for computing $\sum_{i=0}^{m-1} \hat{a}_i \hat{b}_i$, offers similar improvements with respect to DotPGFL, the generic variant based on SumGFL. PolySFL evaluates a polynomial $\sum_{i=0}^{m} \hat{a}_i \hat{x}^i$ by efficiently computing $\{\hat{y}_i\}_{i=1}^{m} = \{\hat{x}^i\}_{i=1}^{m}$ using PowAllFL and then $\sum_{i=0}^{m} \hat{a}_i \hat{y}_i$ using the optimized algorithm for dot products. This combination offers substantial improvements over PolySGFL.

We tested the correctness and performance of the new protocols using our Java implementation of the secure computation framework discussed in Section 2. Table 3 shows the results of a preliminary performance evaluation, for 3 parties, $\ell = 32$, $g = 10$, and $\lceil \log q \rceil = 128$ bits. The computation is carried out on 3 computers with 3.6 GHz CPU connected to an Ethernet LAN with 1 Gbps data rate.

The protocols run a distributed computation with batches of interactive operations executed in parallel, so the running time is strongly affected by the execution environment and implementation optimizations. Table 3 shows the baseline performance of a single-thread implementation and a network with high bandwidth and (very) low latency. For larger batches, the running time can be substantially reduced by optimizing the im-

plementation to efficiently use multiple CPU cores and networks with large bandwidth-delay product. On the other hand, longer network latency means longer interaction rounds.

The running time for sums, dot products, and polynomials is shown for $m \in \{4, 8, 16, 32\}$. SumFL and DotPFL are clearly faster than the generic constructions SumGFL and DotPGFL and the advantage grows with m. PolySFL offers similar improvements over the generic protocol PolySGFL. In these tests, the benefits of low round complexity are attenuated by the short LAN delay. In real-life deployments, with longer network delay, the optimized protocols offer larger performance gains. A more comprehensive performance analysis, with broader scope, will be provided in future work.

REFERENCES

1. M. ALIASGARI, M. BLANTON, F. BAYATBABOLGHANI, *Secure computation of hidden Markov models and secure floating-point arithmetic in the malicious model*, International Journal of Information Security, **16**, *6*, pp. 577–601, 2017.

2. M. ALIASGARI, M. BLANTON, Y. ZHANG, A. STEELE, *Secure computation on floating point numbers*, 20th Annual Network and Distributed System Security Symposium (NDSS13), 2013.

3. D. BOGDANOV, L. KAMM, S. LAUR, V. SOKK, *Rmind: a tool for cryptographically secure statistical analysis*, IEEE Transactions On Dependable And Secure Computing, **15**, *3*, pp. 481–495, 2018.

4. D. BOGDANOV, M. NIITSOO, T. TOFT, J. WILLEMSON, *High-performance secure multi-party computation for data mining applications*, International Journal of Information Security, **11**, *6*, pp. 403–418, 2012.

5. O. CATRINA, *Round-efficient protocols for secure multiparty fixed-point arithmetic*, 12th International Conference on Communications (COMM2018), pp. 431–436, IEEE, 2018.

6. O. CATRINA, *Efficient secure floating-point arithmetic using Shamir secret sharing*, 16th International Joint Conference on e-Business and Telecommunications - volume 2: SECRYPT (Security and Cryptography), pp. 49–60, SciTePress, 2019.

7. O. CATRINA, S. DE HOOGH, *Improved primitives for secure multiparty integer computation*, Security and Cryptography for Networks, Lecture Notes in Computer Science, vol. 6280, pp. 182–199, Springer, 2010.

8. O. CATRINA, S. DE HOOGH, *Secure multiparty linear programming using fixed-point arithmetic*, Computer Security - ESORICS 2010, Lecture Notes in Computer Science, vol. 6345, pp. 134–150, Springer, 2010.

9. O. CATRINA, A. SAXENA, *Secure computation with fixed-point numbers*, Financial Cryptography and Data Security, Lecture Notes in Computer Science, vol. 6052, pp. 35–50. Springer, 2010.

10. R. CRAMER, I. DAMGÅRD, Y. ISHAI, *Share conversion, pseudorandom secret-sharing and applications to secure computation*, Theory of Cryptography (TCC'05), Lecture Notes in Computer Science, vol. 3378, pp. 342–362, Springer, 2005.

11. R. CRAMER, I. B. DAMGÅRD, J. B. NIELSEN, *Secure multiparty computation and secret sharing*, Cambridge University Press, 2015.

12. I. DAMGÅRD, M. FITZI, E. KILTZ, J. NIELSEN, T. TOFT, *Unconditionally secure constant rounds multi-party computation for equality, comparison, bits and exponentiation*, Theory of Cryptography (TCC'06), Lecture Notes in Computer Science, vol. 3876, pp. 285–304, Springer, 2006.

13. I. DAMGÅRD, R. THORBEK, *Non-interactive proofs for integer multiplication*, EUROCRYPT 2007, Lecture Notes in Computer Science, vol. 4515, pp. 412–429, Springer, 2007.

14. V. DIMITROV, L. KERIK, T. KRIPS, J. RANDMETS, J. WILLEMSON, *Alternative implementations of secure real numbers*, 23rd ACM Conference on Computer and Communications Security (CCS'16), pp. 553–564, ACM, 2016.

15. L. KAMM, J. WILLEMSON, *Secure floating point arithmetic and private satellite collision analysis*, International Journal of Information Security, **14**, *6*, pp. 531–548, 2015.

16. D. E. KNUTH, *The art of computer programming, volume 2 (3rd ed.): Seminumerical algorithms*, Addison-Wesley, Boston, MA, USA, 1997.

Received June 22, 2019