Round-Efficient Protocols for Secure Multiparty Fixed-Point Arithmetic

Octavian Catrina
University Politehnica of Bucharest, Bucharest, Romania
E-mail: octavian.catrina(at)elcom.pub.ro

***Abstract*—Secure multiparty computation provides specially designed cryptographic protocols for collaborative applications with private inputs and outputs. Computing with private data has been successfully demonstrated for some real-life applications. However, meeting the application requirements still requires major improvements of the protocols' performance and functionality. In this paper, we present improved building blocks and protocols for secure fixed-point arithmetic based on the framework introduced in [1], [2]. We show that the number of interaction rounds can be substantially reduced with respect to the initial solutions (e.g., multiplication in 1 round, instead of 2 rounds, division in 14 rounds, instead of 43 rounds). The new building blocks used in these protocols are of independent interest, e.g., they can provide much needed improvements for secure floating-point arithmetic.**

***Index Terms*—secure multiparty computation protocols, secure fixed-point arithmetic, secret sharing.**

I. Introduction

Secure computation enables mutually distrustful parties to run joint computations with private inputs. Suppose, for example, that the parties $P_1, P_2, \ldots, P_n$ want to evaluate a function $f(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$, where P_i submits the private input x_i and expects the output y_i. Secure computation provides specially crafted cryptographic protocols that enable the parties to evaluate the function without having to reveal their inputs. The protocols emulate an idealized setting where an incorruptible third party collects the inputs, performs the computation, delivers the results, and then forgets everything.

The overhead of the cryptographic protocols creates a large performance gap between secure computation and traditional computation. When moving to the realm of secure computation, trivial operations often become surprisingly complicated, requiring different approaches and changing the complexity assumptions used in traditional algorithm design. Solutions for various building blocks and applications have gradually been introduced and improved (e.g., [2], [3], [12], [13]). However, meeting the functional, security, and performance requirements of real-life applications is still a challenge.

The goal of this paper is to improve the framework for secure fixed-point computation introduced in [1], [2], by providing new building blocks and arithmetic protocols, whose complexity, in terms of interaction rounds, is substantially reduced (e.g., computation of the binary representation in 3 rounds, multiplication in 1 round, division in 14 rounds).

The practical relevance of the framework was first demonstrated by solving linear programming problems with secret inputs and outputs, for a realistic case study, in the context of collaborative supply-chain optimization [3], [4]. Research published in the mean time presented various other applications and/or further extensions, e.g., secure computation with floating-point numbers [11], [12]. All this related work can benefit from the improved solutions presented in this paper.

Our persistent interest in secure computation with fixed-point numbers, instead of moving on to floating-point numbers, is motivated by two important facts: it is sufficient for a broad range of applications and performs the most frequent operations much more efficiently. For fixed-point numbers, secure addition and subtraction are computed without interaction, while multiplication and inner (vector) product need a single on-line interaction; only secure division is relatively complex. For floating-point numbers, secure addition and subtraction are more complex than fixed-point division.

The paper is structured as follows. Section II briefly reviews the framework, including the secure computation model and the data encoding methods. Section III presents the improved family of protocols for secure fixed-point arithmetic and Section IV describes the building blocks used for secure division. We summarize the main results in Section V.

II. Preliminaries

A. Secure Computation Framework

The protocols described in the next sections rely on abstract primitives that can be instantiated using secret sharing [5] or homomorphic encryption [6]. We focus on solutions based on secret sharing, in the semi-honest model, which are more efficient and more likely to satisfy the performance requirements of practical applications. Support for stronger adversary models can be added by using standard techniques, albeit with a substantial, inherent performance degradation.

Multiparty computation using secret sharing can be summarized as follows. Assume a group of $n > 2$ parties, $P_1, \ldots, P_n$, that communicate on secure channels and run a computation where party P_i has private input x_i and expects output y_i. The parties use a linear secret sharing scheme to create a distributed state of the computation where each party has a share of each secret variable. Then, they use secure computation protocols to compute with shared variables and control the access to their values. The sub-tasks compute with shared inputs and outputs, and thus enable secure protocol composition.

978-1-5386-2350-3/18/$31.00 ©2018 IEEE

The protocols offer perfect or statistical privacy: this means that the views of protocol executions (all values seen by an adversary) can be simulated such that the distributions of real and simulated views are perfectly or statistically indistinguishable, respectively. Let X and Y be distributions with finite sample spaces V and W. The statistical distance between X and Y is $\Delta(X,Y) = \frac{1}{2}\sum_{v \in V \bigcup W} |Pr(X = v) - Pr(Y = v)|$. We say that the distributions are perfectly indistinguishable if $\Delta(X,Y) = 0$ and statistically indistinguishable if $\Delta(X,Y)$ is negligible in some security parameter.

The basic framework uses Shamir secret sharing over a finite field $\mathbb{F}$ (Lagrange polynomial interpolation). It provides secure arithmetic in $\mathbb{F}$ with perfect privacy against a passive threshold adversary able to corrupt t out of n parties. In this model, the parties do not deviate from the protocol and any $t+1$ parties can reconstruct a secret, while t or less parties cannot distinguish it from random values in $\mathbb{F}$. We assume $|\mathbb{F}| > n$, to enable Shamir sharing, and $n > 2t$, for multiplication of secret-shared values. We denote $[\![x]\!]$ a Shamir sharing of x. We refer the reader to [5] for a more formal and general presentation of this approach to secure computation.

The parties can locally compute addition and subtraction of shared field elements, by adding/subtracting their own shares. The other operations are distributed computations, carried out according to dedicated protocols. A notable advantage of Shamir sharing is an efficient secure multiplication protocol.

The protocols discussed in this paper use the field of integers modulo a prime q, denoted $\mathbb{Z}_q$. However, binary computations can be optimized by working in a small field $\mathbb{F}_{2^8}$ [1], [2].

To overcome the limitations of secure arithmetic with shared field elements, many protocols combine secret sharing with additive or multiplicative hiding: given a shared variable $[\![x]\!]$ the parties jointly generate a secret random value $[\![r]\!]$, compute $[\![y]\!] = [\![x]\!] + [\![r]\!]$, and reveal $y = x + r$ (similar to one-time pad encryption of x with key r). For secret $x \in \mathbb{Z}_q$ and random uniform $r \in \mathbb{Z}_q$ we obtain $\Delta(x + r \bmod q, r) = 0$, hence perfect privacy. Alternatively, for $x \in [0, 2^k - 1]$, random uniform $r \in [0, 2^{k+\kappa} - 1]$, and $q > 2^{k+\kappa+1}$ we obtain $\Delta(x + r \bmod q, r) < 2^{-\kappa}$, hence statistical privacy with security parameter κ. Solutions with statistical privacy can substantially simplify the protocols by avoiding wraparound modulo q, although they require larger q for a given data range.

We use two complexity metrics that reflect different aspects of the interaction between parties. Communication complexity measures the amount of data sent by each party. For our protocols, a suitable abstract metric is the number of invocations of 3 primitives during which every party sends a share (field element) to the others: input sharing, multiplication, and secret reconstruction. Round complexity is the number of sequential invocations and is relevant for the network delay, independent of the amount of data. Interactive operations that can be executed in parallel count as a single round. Table I shows the complexity of the core protocols used in this paper.

The protocols are designed assuming implementations that apply several basic optimizations, with major impact on performance. Interactive operations that do not depend on each other are executed in parallel, in a single round. In particular, all shared random values can be precomputed in parallel. We use Pseudo-random Replicated Secret Sharing (PRSS) [7] and its integer variant (RISS) [8] to generate without interaction shared random field elements and integers, and random sharings of 0. Still, for some shared random values, we cannot avoid interaction (e.g., random bits shared in $\mathbb{Z}_q$). To better emphasize the actual on-line complexity, we indicate separately the complexity of the precomputation round.

TABLE I
CORE PROTOCOLS (SELECTION).

Sharing and arithmetic with field elements	Rounds	Inter.op. ($\mathbb{Z}_q$)
$[\![a]\!] \leftarrow \mathsf{Share}(a)$	1	1
$a \leftarrow \mathsf{Reveal}([\![a]\!])$	1	1
$[\![c]\!] \leftarrow [\![a]\!] + [\![b]\!]$	0	0
$[\![c]\!] \leftarrow [\![a]\!][\![b]\!]$	1	1
$[\![c]\!] \leftarrow [\![a]\!]b$; $[\![c]\!] \leftarrow [\![a]\!] + b$	0	0
$[\![c]\!] \leftarrow \mathsf{InnerProd}(\{[\![a_i]\!]\}_{i=1}^m, \{[\![b_i]\!]\}_{i=1}^m)$	1	1
Joint generation of shared random values	**Rounds**	**Inter.op. ($\mathbb{Z}_q$)**
$[\![r]\!] \leftarrow \mathsf{PRandFld}()$	0	0
$[\![r]\!] \leftarrow \mathsf{PRandBit}()$	1	1
$[\![r]\!] \leftarrow \mathsf{PRandInt}(k)$	0	0
$[\![0]\!]_{2u} \leftarrow \mathsf{PRZS}(u)$	0	0
$[\![r'']\!], [\![r']\!], \{[\![r'_i]\!]\}_{i=0}^{m-1} \leftarrow \mathsf{PRandM}(k,m)$	1	m

B. Data Encoding for Secure Computation

In this paper, we focus on secure computation with the following data types: binary values, signed integers $\mathbb{Z}_{\langle k \rangle} = \{\bar{x} \in \mathbb{Z} | -2^{k-1} \leq \bar{x} \leq 2^{k-1} - 1\}$ and signed fixed-point numbers $\mathbb{Q}_{\langle k,f \rangle} = \{\tilde{x} \in \mathbb{Q} | \tilde{x} = \bar{x} \cdot 2^{-f}, \bar{x} \in \mathbb{Z}_{\langle k \rangle}\}$. Intuitively, $\mathbb{Q}_{\langle k,f \rangle}$ is obtained by sampling the range of real numbers $[-2^{k-f-1}, 2^{k-f-1} - 2^{-f}]$ at 2^{-f} intervals; we call the value 2^{-f} the resolution of the fixed-point numbers. The parameters k and f are not secret. For secure computation, these data types are encoded in a field $\mathbb{F}$ as follows [2].

Logical values $false, true$ and bit values $0, 1$ are encoded as 0_F and 1_F, respectively. $\mathbb{F}$ can be either $\mathbb{Z}_q$ or a small binary field $\mathbb{F}_{2^m}$. This encoding allows efficient secure evaluation of Boolean functions using secure arithmetic in $\mathbb{F}$ [1].

Signed integers $\bar{x} \in \mathbb{Z}_{\langle k \rangle}$ are efficiently 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 a prime $q > 2^k$ (similar to two's complement encoding). This encoding enables efficient secure integer arithmetic using 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 have $\bar{x}_1 \odot \bar{x}_2 = \mathsf{fld}^{-1}(\mathsf{fld}(\bar{x}_1) \odot \mathsf{fld}(\bar{x}_2))$; 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 numbers $\tilde{x} \in \mathbb{Q}_{\langle k,f \rangle}$ are mapped to $\mathbb{Z}_{\langle k \rangle}$ by the function $\mathsf{int} : \mathbb{Q}_{\langle k,f \rangle} \mapsto \mathbb{Z}_{\langle k \rangle}$, $\bar{x} = \mathsf{int}_f(\tilde{x}) = \tilde{x} 2^f$, and then encoded in $\mathbb{Z}_q$ as described above.

Using q's complement encoding instead of sign-and-magnitude simplifies considerably the secure computation: the operations with signed numbers translate naturally to the corresponding operations with shared field elements, regardless of the sign, and the compact encoding reduces the communication overhead. Moreover, the protocols for computing with integers can also be used or easily adapted for fixed-point rational numbers (e.g., addition, subtraction and comparison).

III. SECURE FIXED-POINT ARITHMETIC

We focus in the following on secure computation using a field $\mathbb{Z}_q$, with a prime $q > 2^{2k+\kappa}$ (the protocol for generating

TABLE II
MOD-2^m ARITHMETIC AND COMPARISON (SELECTION).

Protocol	Rounds	Inter.op. ($\mathbb{Z}_q$)	Prec. ($\mathbb{Z}_q$)
$\mathsf{Div2mP}(\llbracket a \rrbracket, k, m)$	1	1	m
$\mathsf{Div2m}(\llbracket a \rrbracket, k, m)$	3	$m+2$	$3m-1$
$\mathsf{Mod2m}(\llbracket a \rrbracket, k, m)$	3	$m+2$	$3m-1$
$\mathsf{Mod2}(\llbracket a \rrbracket, k)$	1	1	1
$\mathsf{BitLT}(\llbracket a \rrbracket, b)$	2	$k+1$	$2k$
$\mathsf{LTZ}(\llbracket a \rrbracket, k)$	3	$k+1$	$3k-3$
$\mathsf{EQZ}(\llbracket a \rrbracket, k)$	3	$\log(k)+2$	$k+3\log(k)-2$

shared random bits also requires $q \bmod 4 = 3$). This allows us to encode signed numbers with up to $2k$ bits and provide statistical privacy with the security parameter κ.

a) Scaling: The position of the radix point is not actually fixed throughout the computation. In fact, we often need to change the resolution. For $\tilde{a} \in \mathbb{Q}_{\langle k,f \rangle}$ we obtain the encoding of $\tilde{a}$ with resolution $f+f_1$ by computing $\bar{a}' = \bar{a} \cdot 2^{f_1}$. Similarly, we obtain an approximation of $\tilde{a}$ with resolution $f - f_1$ by computing $\tilde{a}' = \bar{a}/2^{f_1}$. Multiplication by a non-secret integer can be achieved without interaction. However, division by 2^m (truncation) is more problematic.

We briefly review the protocols for division by 2^m presented in [1]. For a secret signed integer $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$ and a public integer $m \in [1, k-1]$, Protocol 1, Div2mP computes $\bar{a}/2^m$ with probabilistic rounding to nearest. The result is the nearest integer with probability $1-\alpha$, where α is the distance between $\bar{a}/2^m$ and that integer.

Protocol 1: $\llbracket c \rrbracket \leftarrow \mathsf{Div2mP}(\llbracket a \rrbracket, k, m)$

1 $(\llbracket r'' \rrbracket, \llbracket r' \rrbracket, \{\llbracket r'_i \rrbracket\}_{i=1}^m) \leftarrow \mathsf{PRandM}(k, m)$;
2 $\llbracket r \rrbracket \leftarrow 2^m \llbracket r'' \rrbracket + \llbracket r' \rrbracket$;
3 $b \leftarrow \mathsf{Reveal}(2^{k-1} + \llbracket a \rrbracket + \llbracket r \rrbracket)$;
4 $b' \leftarrow b \bmod 2^m$;
5 $\llbracket c \rrbracket \leftarrow (\llbracket a \rrbracket - (b' - \llbracket r' \rrbracket))2^{-m}$;
6 return $\llbracket c \rrbracket$;

Let $\bar{d} = 2^{k-1} + \bar{a}$ and $\bar{a}' = \bar{a} \bmod 2^m$. Observe that $\bar{d} \bmod 2^m = \bar{a}'$ for any $m \in [1, k-1]$. The protocol reveals $b = d + r$, where $r = 2^m r'' + r'$ is a random secret integer that hides d with statistical secrecy and $r' = \sum_{i=1}^m 2^i r_i$, with $\{r'_i\}_{i=1}^m$ uniformly random secret bits. Observe that $b' = (d + r) \bmod 2^m = a' + r' - 2^m u$, where $u = ((b' < r')?\ 1 : 0)$. Therefore, step 5 computes $\bar{c} = (\bar{a} - \bar{a}' + 2^m u)2^{-m} = \lfloor \bar{a}/2^m \rfloor + u$.

A variant with deterministic rounding to $-\infty$ is obtained by computing the bit u, using the protocol BitLT. We structure the computation into two building blocks: Protocol 2, Mod2m, computes $\bar{a} \bmod 2^m$; Protocol 3, Div2m computes $\lfloor \bar{a}/2^m \rfloor$ using Mod2m.

Protocol 2: $\llbracket c \rrbracket \leftarrow \mathsf{Mod2m}(\llbracket a \rrbracket, k, m)$

1 $(\llbracket r'' \rrbracket, \llbracket r' \rrbracket, \{\llbracket r'_i \rrbracket\}_{i=1}^m) \leftarrow \mathsf{PRandM}(k, m)$;
2 $b \leftarrow \mathsf{Reveal}(2^{k-1} + \llbracket a \rrbracket + 2^m \llbracket r'' \rrbracket + \llbracket r' \rrbracket)$;
3 $b' \leftarrow b \bmod 2^m$;
4 $\llbracket u \rrbracket \leftarrow \mathsf{BitLT}(b', \{\llbracket r'_i \rrbracket\}_{i=1}^m)$;
5 $\llbracket c \rrbracket \leftarrow b' - \llbracket r' \rrbracket + 2^m \llbracket u \rrbracket$;
6 return $\llbracket c \rrbracket$;

Protocol 3: $\llbracket c \rrbracket \leftarrow \mathsf{Div2m}(\llbracket a \rrbracket, k, m)$

1 $\llbracket b \rrbracket \leftarrow \mathsf{Mod2m}(\llbracket a \rrbracket, k, m)$;
2 $\llbracket c \rrbracket \leftarrow (\llbracket a \rrbracket - \llbracket b \rrbracket)2^{-m}$;
3 return $\llbracket c \rrbracket$;

The complexity of these protocols is shown in Table II (round efficient variants in [1]). Observe that Div2mP is much more efficient than Div2m. $\mathsf{LTZ}(\llbracket a \rrbracket, k)$ computes $s = (\bar{a} < 0)?1 : 0$ as $\llbracket s \rrbracket = -\mathsf{Div2m}(\llbracket a \rrbracket, k, k-1)$.

Protocol 4, BitLT [1] takes on input a non-secret integer $a = \sum_{i=1}^k 2^i a_i$ and a bitwise shared integer $b = \sum_{i=1}^k 2^i b_i$ and computes the secret bit $u = (a < b)?\ 1 : 0$.

Protocol 4: $\llbracket u \rrbracket \leftarrow \mathsf{BitLT}(a, \{\llbracket b_i \rrbracket\}_{i=1}^k)$

1 foreach $i \in [1, k]$ do
2 $\quad \llbracket d_i \rrbracket \leftarrow a_i + \llbracket b_i \rrbracket - 2a_i \llbracket b_i \rrbracket$;
3 $\{\llbracket p_i \rrbracket\}_{i=1}^k \leftarrow \mathsf{SufMul}(\{\llbracket d_i + 1 \rrbracket\}_{i=1}^k)$;
4 $\llbracket s \rrbracket \leftarrow (1 - a_k)\llbracket d_k \rrbracket + \sum_{i=1}^{k-1}(1 - a_i)(\llbracket p_i \rrbracket - \llbracket p_{i+1} \rrbracket)$;
5 $\llbracket u \rrbracket \leftarrow \mathsf{Mod2}(\llbracket s \rrbracket, k)$;
6 return $\llbracket u \rrbracket$;

Steps 1-3 compute $d_i = \mathsf{XOR}(a_i, b_i)$ and the products $p_i = \prod_{j=i}^k (d_j + 1)$, for $i \in [1, k]$; the protocol SufMul is identical to PreMul in [1], with the inputs and outputs in inverse order. Observe that $d_j + 1 = 2^{d_j}$ and so $p_i = \prod_{j=i}^k 2^{d_j} = 2^{\sum_{j=i}^k d_j}$. Step 4 computes $s = (1 - a_k)d_k + \sum_{i=1}^{k-1}(1 - a_i)(p_i - p_{i+1})$. Since $p_i - p_{i+1} = d_i p_{i+1}$, it follows that $s = (1 - a_k)d_k + \sum_{i=1}^{k-1}(1 - a_i)d_i 2^{\sum_{j=i+1}^k d_j}$. Assume $a \neq b$ and let $m \leq k$ be the secret index of the most significant different bit. The expected result is $u = 1 - a_m$. If $m < k$, $d_m = 1$ and $d_i = 0$ for all $i \in [m+1, k]$, hence $s = 1 - a_m + 2(1 - a_{m-1})d_{m-1} + 2\sum_{i=1}^{m-2}(1 - a_i)d_i 2^{\sum_{j=i+1}^{m-1} d_j}$. Step 5 computes $u = s \bmod 2 = 1 - a_m$, the correct result. If $m = k$, $d_k = 1$ and $s = 1 - a_k + 2(1 - a_{k-1})d_{k-1} + 2\sum_{i=1}^{k-2}(1 - a_i)d_i 2^{\sum_{j=i+1}^{k-1} d_j}$. The computed result is $u = s \bmod 2 = 1 - a_k$. Finally, if $a = b$, the expected result is $u = 0$. In this case, $d_i = 0$ for all $i \in [1, k]$. The protocol computes $s = 0$ and hence $u = 0$.

b) Addition, subtraction and comparison: Let $\tilde{a}_1 \in \mathbb{Q}_{\langle k,f_1 \rangle}$ and $\tilde{a}_2 \in \mathbb{Q}_{\langle k,f_2 \rangle}$ with integer representations $\bar{a}_1 = \tilde{a}_1 2^{f_1}$ and $\bar{a}_2 = \tilde{a}_2 2^{f_2}$. If $f_1 = f_2$ secure addition, subtraction and comparison are computed using the same protocols as for integers. Otherwise, we have to scale one operand, so that both have the same resolution. For example, if $f_2 > f_1$ we obtain $\tilde{a}_3 = \tilde{a}_1 + \tilde{a}_2$ with resolution 2^{-f_2} by computing $\bar{a}'_1 = 2^{f_2 - f_1}\bar{a}_1$ and $\bar{a}_3 = \bar{a}'_1 + \bar{a}_2$. Note that addition and subtraction are computed without interaction and the result is exact. This is an important advantage of secure fixed-point arithmetic over floating-point arithmetic.

c) Multiplication: Suppose we want to compute $\tilde{a}_3 = \tilde{a}_1 \tilde{a}_2$. Observe that $\tilde{a}_1 \tilde{a}_2 = \bar{a}_1 \bar{a}_2 2^{-(f_1+f_2)} \in \mathbb{Q}_{\langle 2k, f_1+f_2 \rangle}$. This is the exact value of $\tilde{a}_3$, with resolution $2^{-(f_1+f_2)}$. Therefore, secure fixed-point multiplication can be efficiently computed with a secure integer multiplication. Moreover, the fixed-point types of the inputs may be different. In particular, an input may be a fixed-point type and the other input an integer type.

This simple solution has limited applicability, because each multiplication increases the fractional part. Typical fixed-point

multiplications consist of an integer multiplication followed by scaling, to shorten the fractional part. This is shown in Protocol 5, FXMul: for secret inputs $\tilde{a}_1, \tilde{a}_2 \in \mathbb{Q}_{\langle k,f\rangle}$, the protocol computes secret $\tilde{a}_3 \approx \bar{a}_3 2^{-f}$, where $\bar{a}_3 = \bar{a}_1\bar{a}_2 2^{-f}$. The absolute error due to the truncation is bound by 2^{-f}.

Protocol 5: $[\![a_3]\!] \leftarrow \mathsf{FXMul}([\![a_1]\!], [\![a_2]\!], k, f)$

1 $[\![a]\!] \leftarrow [\![a_1]\!][\![a_2]\!]$;
2 $[\![a_3]\!] \leftarrow \mathsf{Div2mP}([\![a]\!], k+f, f)$; // or Div2m
3 return $[\![a_3]\!]$;

Using Div2mP, the protocol FXMul runs in 2 rounds with 2 interactive operations. However, we can further improve it, as described in the following.

The optimization applies to any multiplication followed by additive hiding: $[\![c]\!] \leftarrow [\![a]\!][\![b]\!]; d \leftarrow \mathsf{Reveal}([\![c]\!] + [\![r]\!])$. We denote $[\![x]\!]_{u,i}$ the share of x owned by party i, for a random polynomial of degree u; to simplify the notation, we omit the degree when it takes the default value t.

The parties compute $[\![c]\!] \leftarrow [\![a]\!][\![b]\!]$ as follows: each party $i \in [1,n]$ locally computes the shares $[\![c]\!]_{2t,i} \leftarrow [\![a]\!]_i[\![b]\!]_i + [\![z]\!]_{2t,i}$, where $[\![z]\!]_{2t,i}$ are pseudo-random shares of zero generated by $\mathsf{PRZS}(2t)$. Note that $[\![a]\!]_i[\![b]\!]_i$ are shares of c for a polynomial of degree $2t$, which is the product of the polynomials used to share a and b. As the sharing polynomial is not random, these share products cannot be revealed. The problem is solved by adding random shares of 0, locally computed by PRZS.

We denote $[\![a_1]\!] * [\![a_2]\!]$ the local computation of random shares of a_1a_2, for a polynomial of degree $2t$, as described above. The parties can then reveal $[\![c]\!]_{2t} + [\![r]\!]$ using RevealD, the variant of the secret reconstruction protocol for degree $2t$. This eliminates the interaction required by the classical protocol for multiplication of Shamir-shared field elements.

The optimization applies to all the protocols in Table II, since they all start with additive hiding of the input (except the underlying building block BitLT). We can define, therefore, variants of all these protocols for input shared with threshold $2t$, and distinguish them by the suffix "D". The only difference is that they use RevealD instead of Reveal.

The optimized fixed-point multiplication is shown in Protocol 6, FXMulD. With Div2mPD, it needs a single interaction. With Div2mD, it needs 3 rounds (instead of 4 rounds with Div2m), but probabilistic rounding to nearest is suitable for most practical applications. Note also that the inner product of two vectors of arbitrary length, with secret fixed-point inputs and output, is computed with the same complexity as a single multiplication; this implies very efficient matrix multiplication.

Protocol 6: $[\![a_3]\!] \leftarrow \mathsf{FXMulD}([\![a_1]\!], [\![a_2]\!], k, f)$

1 $[\![a_3]\!] \leftarrow \mathsf{Div2mPD}([\![a_1]\!] * [\![a_2]\!], k+f, f)$; // or Div2mD
2 return $[\![a_3]\!]$;

d) Division: Finally, we want a protocol that computes $\tilde{c} = \tilde{a}/\tilde{b}$, for $\tilde{a}, \tilde{b} \in \mathbb{Q}_{\langle k,f\rangle}$. If $\tilde{a}$ is secret and $\tilde{b}$ is public, we can efficiently compute $\tilde{c} = \tilde{a}/\tilde{b}$ with a single interaction: $[\![c]\!] \leftarrow \mathsf{Div2mP}([\![a]\!] \cdot \mathsf{fld}(\mathsf{int}_f(1/\tilde{b})), k+f, f)$. On the other hand, if $\tilde{b}$ is secret, secure division becomes expensive.

We describe in the following an improved version of the division protocol introduced in [2], based on Goldschmidt's algorithm [14]. Let $a, b \in \mathbb{R}$, $b \neq 0$. Goldschmidt's algorithm starts with an initial approximation $w_0 \approx 1/b$, with relative error $\epsilon_0 < 1$, and computes a/b iteratively, as follows: let $c_1 = aw_0$, $d_1 = \epsilon_0 = 1 - bw_0$; for $i \geq 1$ do $c_{i+1} = c_i(1+d_i)$, $d_{i+1} = d_i^2$. After i iterations we obtain $c_{i+1} = (a/b)(1-\epsilon_0^{2^i})$, so $c_{i+1} \approx a/b$ with relative error $\epsilon_0^{2^i}$.

The main difficulty is to obtain an initial approximation that ensures fast convergence. We use the following method [15]: we compute $\bar{v}$ and $\tilde{b}' = \tilde{b}\bar{v}$, so that $|\tilde{b}'| \in [0.5, 1]$; we say that $\tilde{b}'$ is the normalized value of $\tilde{b}$; then, we compute $\tilde{w}_0' = 2.9142 - 2\tilde{b}'$, a linear approximation of $1/\tilde{b}'$ with relative error $\epsilon_0 < 0.08578$, and $\tilde{w}_0 = \tilde{w}_0'\bar{v}$, the approximation of $1/\tilde{b}$. This approximation provides about 3.5 exact bits, so for a k-bit input the protocol needs $\theta = \lceil \log_2 \frac{k}{3.5} \rceil$ iterations. Moreover, the protocol can compute it without interaction.

Protocol 7, FXDiv, computes $\tilde{c} = \tilde{a}/\tilde{b}$, for $\tilde{a}, \tilde{b} \in \mathbb{Q}_{\langle k,f\rangle}$, $\tilde{b} \neq 0$, with secret inputs and outputs, using the algorithm described above. The error flag $z = (\tilde{b} = 0)?\ 1 : 0$ indicates an attempt to divide by 0 (i.e., the output is valid if $z = 0$).

Protocol 7: $([\![c]\!], [\![z]\!]) \leftarrow \mathsf{FXDiv}([\![a]\!], [\![b]\!], k, f)$

1 $\theta \leftarrow \lceil \log_2 \frac{k}{3.5} \rceil$; $\alpha \leftarrow \mathsf{fld}(\mathsf{int}_{2f}(1.0))$;
2 $([\![w]\!], [\![z]\!]) \leftarrow \mathsf{AppRec}([\![b]\!], k)$;
3 $[\![c]\!] \leftarrow \mathsf{Div2mPD}([\![a]\!] * [\![w]\!], 2k, f)$;
4 $[\![d]\!] \leftarrow \alpha - [\![b]\!][\![w]\!]$;
5 foreach $i \in [1, \theta - 1]$ do
6 $\quad [\![c]\!] \leftarrow \mathsf{Div2mPD}([\![c]\!] * (\alpha + [\![d]\!]), 2k, 2f)$;
7 $\quad [\![d]\!] \leftarrow \mathsf{Div2mPD}([\![d]\!] * [\![d]\!], 2k, 2f)$;
8 $[\![c]\!] \leftarrow \mathsf{Div2mPD}([\![c]\!] * (\alpha + [\![d]\!]), 2k, 2f)$;
9 return $([\![c]\!], [\![z]\!])$;

FXDiv starts by computing the initial approximation $\tilde{w} \approx 1/\tilde{b}$ using the protocol AppRec. If $\tilde{b} > 0$, AppRec computes $\tilde{b}' \in [0.5, 1]$ and the linear approximation $\tilde{w}' = 2.9142 - 2\tilde{b}' \approx 1/\tilde{b}'$. Similarly, if $\tilde{b} < 0$, it computes $\tilde{b}' \in [-1, -0.5]$ and the approximation $\tilde{w}' = -2.9142 - 2\tilde{b}'$. Steps 3-8 compute $\tilde{c} = \tilde{a}/\tilde{b}$ using Goldschmidt's algorithm. To improve the accuracy, the bit-length of the fractional part of the error variable d is set to $2f$ bits. The multiplications in each iteration are computed in parallel, in a single round.

Given a secret fixed-point number $\tilde{b} \in \mathbb{Q}_{\langle k,f\rangle}$, Protocol 8, AppRec, computes the initial approximation $\tilde{w} \approx 1/\tilde{b}$, $\tilde{w} \in \mathbb{Q}_{\langle k,f\rangle}$, and the flag $z = (\tilde{b} = 0)?\ 1 : 0$, with secret outputs.

Protocol 8: $([\![w]\!], [\![z]\!]) \leftarrow \mathsf{AppRec}([\![b]\!], k, f)$

1 $\alpha \leftarrow \mathsf{fld}(\mathsf{int}_{k-1}(2.9142))$;
2 $\{[\![b_i]\!]\}_{i=0}^{k-1} \leftarrow \mathsf{BitDec}([\![b]\!], k, k)$;
3 foreach $i \in [0, k-2]$ do
4 $\quad [\![b_i']\!] \leftarrow [\![b_i]\!] + [\![b_{k-1}]\!] - 2[\![b_i]\!][\![b_{k-1}]\!]$;
5 $\{[\![c_i]\!]\}_{i=0}^{k-2} \leftarrow \mathsf{SufOr}(\{[\![b_i']\!]\}_{i=0}^{k-2})$;
6 $[\![v]\!] \leftarrow 1 + \sum_{i=0}^{k-2} 2^i(1 - [\![c_i]\!])$;
7 $[\![z]\!] \leftarrow 1 - ([\![c_0]\!] \vee [\![b_{k-1}]\!])$;
8 $[\![w']\!] \leftarrow \alpha(1 - 2[\![b_{k-1}]\!]) - 2[\![v]\!][\![b]\!]$;
9 $[\![w]\!] \leftarrow \mathsf{Div2mPD}([\![v]\!] * [\![w']\!], 2k, 2(k-f-1))$;
10 return $([\![w]\!], [\![z]\!])$;

In step 2, AppRec uses the protocol BitDec to compute $\{b_i\}_{i=0}^{k-1}$, the two's complement binary encoding of $\bar{b} = \tilde{b}2^f$.

Note that $b_{k-1} = (\tilde{b} < 0)?\ 1 : 0$ (the sign bit).

Suppose $\bar{b} > 0$ and $\bar{b} \in [2^{m-1}, 2^m - 1]$, for some $m \leq k-1$. In this case, AppRec computes the normalization factor $\bar{v} = 2^{k-m-1}$ so that $\bar{b}' = \bar{v}\bar{b} \in [2^{k-2}, 2^{k-1} - 1]$. Intuitively, the multiplication $\bar{v}\bar{b}$ moves $\bar{b}$'s most significant non-zero bit from index $m-1$ to $k-2$ (after the sign bit). We'll use $\bar{b}' = \bar{v}\bar{b}$ as a fixed-point number with resolution $2^{-(k-1)}$, so $\tilde{b}' = \bar{b}'2^{-(k-1)} \in [0.5, 1]$, as required. If $\bar{b} < 0$, AppRec computes $\bar{v}$ so that $\bar{b}' = \bar{v}\bar{b} \in [-2^{k-1}, -2^{k-2} - 1]$ and $\tilde{b}' = \bar{b}'2^{-(k-1)} \in [-1, -0.5]$. Intuitively, in this case, $\bar{v}\bar{b}$ moves the most significant zero bit to index $k-2$. The two cases are handled obliviously by computing $b'_i = \mathsf{XOR}(b_i, b_{k-1})$, for $i \in [0, k-2]$ in steps 3-4.

Step 5 computes $\{c_i\}_{i=0}^{k-2} = \{\bigvee_{j=i}^{k-2} b_j\}_{i=0}^{k-2}$, using the protocol SufOr (Suffix-OR). SufOr is the protocol PreOr (Prefix-OR) in [1], with the inputs and outputs in inverse order. Observe that $c_i = 0$ for $i \in [m, k-2]$ and $c_i = 1$ for $i \in [0, m-1]$. Therefore, we can compute $v = 1 + \sum_{i=0}^{k-2} 2^i(1 - c_i) = 2^{k-m-1}$. Also, since $c_0 = \bigvee_{j=0}^{k-2} b'_j$, we compute $z = 1 - (c_0 \vee b_{k-1})$ in step 7.

Steps 8 computes the normalized divisor $\tilde{b}' = \bar{v}\tilde{b}$ and the approximation $\tilde{w}' \approx 1/\tilde{b}'$ with resolution $2^{-(k-1)}$. Step 9 scales $\tilde{w}'$ to obtain $\tilde{w} \approx 1/\tilde{b}$ with resolution 2^{-f}. The normalization factor is $\bar{v} = 2^{k-m-1}$, hence $\tilde{b}' = \bar{v}\tilde{b}2^f 2^{-(k-1)} = \tilde{b}2^{f-m}$ and $\tilde{w} = \tilde{w}'2^{f-m} \approx 1/\tilde{b}$. Step 9 computes $\bar{w} = \bar{v}\bar{w}'2^{-2(k-f-1)}$, because $\tilde{w}2^f = \tilde{w}'2^{f-m}2^f = \bar{v}2^{m-(k-1)}\bar{w}'2^{-(k-1)}2^{2f-m} = \bar{v}\bar{w}'2^{-2(k-f-1)}$. Observe that this method avoids division by a secret power of 2, which is much more complex.

Table III summarizes the complexity of FXDiv and its main building blocks. BitDec is presented in the next section.

TABLE III
FIXED-POINT MULTIPLICATION, DIVISION AND BUILDING BLOCKS.

Protocol	Rounds	Inter. op.	Prec. ($k = 2f$)
$\mathsf{FXMulID}([\![a_1]\!], [\![a_2]\!], k, f)$	1	1	f
$\mathsf{FXDiv}([\![a]\!], [\![b]\!], k, f)$	$9 + \theta$	$5k + 2\theta$	$\approx 8k + 2\theta k$
$\mathsf{AppRec}([\![b]\!], k)$	8	$5k$	$8k - 6$
$\mathsf{BitDec}([\![a]\!], k, k)$	3	$2k + 1$	$4k - 1$
$\mathsf{SufOr}(\{[\![a_i]\!]\}_{i=1}^{k})$	2	$2k - 1$	$3k$

For input bit-length $k \in [64, 112]$, FXDiv executes $\theta = 5$ iterations and needs 14 rounds and about $5k$ on-line interactive operations. Moreover, if the sign of the divisor is not secret, steps 3-4 in AppRec are not necessary and the protocol needs only 13 rounds and about $4k$ interactive operations. This is a substantial improvement with respect to the division protocol presented in [2], which needs 43 rounds for secret sign and 33 rounds for non-secret sign. The improvement is due to round-efficient versions of fixed-point multiplication, BitDec, and SufOr, as well as efficient handling of divisors with secret sign (e.g., removal of a secure comparison)[1].

IV. Round-Efficient Building Blocks

We introduce in this section the new, round-efficient building blocks used in FXDiv. Actually, they are general and powerful building blocks, that are also useful for other applications. Their complexity is summarized in Table IV.

TABLE IV
ADDITIONAL ROUND-EFFICIENT BUILDING BLOCKS.

Protocol	Rounds	Inter.op. ($\mathbb{Z}_q$)	Prec. ($\mathbb{Z}_q$)
$\mathsf{BitDec}([\![a]\!], k, m)$	3	$2m + 1$	$4m - 1$
$\mathsf{PreMod2m}([\![a]\!], k, m)$	3	$2m + 1$	$4m - 1$
$\mathsf{PreDiv2m}([\![a]\!], k, m)$	3	$2m + 1$	$4m - 1$
$\mathsf{PreBitLT}(a, \{[\![b_i]\!]\}_{i=1}^{k})$	2	$2k$	$3k - 1$
$\mathsf{PreMulInv}(\{[\![a_i]\!]\}_{i=1}^{k})$	1	k	$2k - 1$

a) Prefix reduction modulo 2^m: Protocol 9, PreMod2m, is a generalization of Mod2m: it takes as inputs a secret integer $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$ and a public integer $m \in [1, k-1]$, and outputs the secret integers $\{\bar{a}'_i\}_{i=1}^m = \{\bar{a} \bmod 2^i\}_{i=1}^m$. PreMod2m uses the protocol PreBitLT, a generalization of BitLT, to efficiently compute the secret bits $\{u_i\}_{i=1}^m = \{(c'_i < r'_i)?\ 1 : 0\}_{i=1}^m$.

Protocol 9: $\{[\![a'_i]\!]\}_{i=1}^m \leftarrow \mathsf{PreMod2m}([\![a]\!], k, m)$

1 $([\![r'']\!], [\![r']\!], \{[\![r'_i]\!]\}_{i=1}^m) \leftarrow \mathsf{PRandM}(k, m)$; // prec.
2 $c \leftarrow \mathsf{Reveal}(2^{k-1} + [\![a]\!] + 2^m[\![r'']\!] + [\![r']\!])$;
3 foreach $i \in [1, m]$ do
4 $\quad c_i \leftarrow c \bmod 2^i$;
5 $\quad [\![s_i]\!] \leftarrow \sum_{j=0}^{i-1} 2^j [\![r'_j]\!]$;
6 $\{[\![u_i]\!]\}_{i=1}^m \leftarrow \mathsf{PreBitLT}(c, \{[\![r'_i]\!]\}_{i=1}^m)$;
7 foreach $i \in [1..m]$ do
8 $\quad [\![a'_i]\!] \leftarrow c_i - [\![s_i]\!] + 2^i[\![u_i]\!]$;
9 return $\{[\![a'_i]\!]\}_{i=1}^m$;

Using PreMod2m, we immediately obtain PreDiv2m, a similar generalization of Div2m, that efficiently computes $\{\bar{a}'_i\}_{i=1}^m = \{\lfloor \bar{a}/2^i \rfloor\}_{i=1}^m$, with secret input and outputs.

b) Binary encoding: Given a secret integer $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$, Protocol 10, BitDec, computes $m \in [1, k]$ least significant secret bits of $\bar{a}$'s two's complement binary encoding.

Protocol 10: $\{[\![a_i]\!]\}_{i=0}^{m-1} \leftarrow \mathsf{BitDec}([\![a]\!], k, m)$

1 $\{[\![b_i]\!]\}_{i=1}^m \leftarrow \mathsf{PreMod2m}([\![a]\!], k, m)$;
2 $[\![a_0]\!] \leftarrow [\![b_1]\!]$;
3 foreach $i \in [1, m-1]$ do
4 $\quad [\![a_i]\!] \leftarrow ([\![b_{i+1}]\!] - [\![b_i]\!])2^{-i}$;
5 return $\{[\![a_i]\!]\}_{i=0}^{m-1}$;

Let $\bar{a} = -2^{k-1}a_{k-1} + \sum_{i=0}^{k-2} 2^i a_i$, with $\{a_i\}_{i=0}^{k-1}$ the bits of the two's complement encoding. Observe that $\bar{a} \bmod 2^k = 2^{k-1}a_{k-1} + \sum_{i=0}^{k-2} 2^i a_i$ and $\bar{a} \bmod 2^i = \sum_{j=0}^{i-1} 2^j a_j$ for all $i \in [1, k-1]$. Protocol PreMod2m computes $\bar{b}_i = \bar{a} \bmod 2^i$, for $i \in [1, m]$, with secret input and outputs. We first obtain $a_0 = \bar{a} \bmod 2 = b_1$. Let $\bar{d}_i = \bar{b}_{i+1} - \bar{b}_i$, for $i \in [1, k-1]$. Observe that $\bar{d}_i = 2^i a_i$. Since 2^i divides $\bar{d}_i$, the parties can compute $a_i = d_i(2^{-i} \bmod q)$ (division in $\mathbb{Z}_q$). Protocol 10 runs in 3 rounds, with about $2m$ on-line interactive operations, providing an efficient alternative to the variant in [2], which needs $\log(m) + 1$ rounds and $m \log m$ on-line operations.

c) Prefix binary comparison: Protocol 11, PreBitLT, is a generalization of BitLT: given a non-secret integer $a = \sum_{i=1}^k 2^i a_i$ and a bitwise shared integer $b = \sum_{i=1}^k 2^i b_i$, it computes the secret bits $\{u_i\}_{i=1}^k = \{(a'_i < b'_i)?\ 1 : 0\}_{i=1}^k$, where $a'_i = \sum_{j=1}^i 2^j a_j$ and $b'_i = \sum_{j=1}^i 2^j b_j$.

[1]The variants of BitDec and PreOr used in [2] trade off a larger number of rounds for lower communication complexity (as discussed in [1]): both need $\log(k)$ rounds, but the computation is carried out in $\mathbb{F}_{2^8}$.

Protocol 11: $\{[\![u_i]\!]\}_{i=1}^k \leftarrow \mathsf{PreBitLT}(a, \{[\![b_i]\!]\}_{i=1}^k)$

1 foreach $i \in [1,k]$ do
2 $[\![d_i]\!] \leftarrow a_i + [\![b_i]\!] - 2a_i[\![b_i]\!]$;
3 $(\{[\![p_i]\!]\}_{i=1}^k, \{[\![p'_i]\!]\}_{i=1}^k) \leftarrow \mathsf{SufMulInv}(\{[\![d_i]\!]+1\}_{i=1}^k)$;
4 $[\![s_1]\!] \leftarrow (1-a_1)([\![p_1]\!] - [\![p_2]\!])$;
5 foreach $i \in [2, k-1]$ do
6 $[\![s_i]\!] \leftarrow [\![s_{i-1}]\!] + (1-a_i)([\![p_i]\!] - [\![p_{i+1}]\!])$;
7 $[\![s_k]\!] \leftarrow [\![s_{k-1}]\!] + (1-a_k)[\![d_k]\!]$;
8 foreach $i \in [1, k-1]$ do parallel
9 $[\![u_i]\!] \leftarrow \mathsf{Mod2D}([\![s_i]\!] * [\![p'_{i+1}]\!], k)$;
10 $[\![u_k]\!] \leftarrow \mathsf{Mod2}([\![s_k]\!], k)$;
11 return $\{[\![u_i]\!]\}_{i=1}^k$;

For $s_k = (a'_k < b'_k)?\ 1 : 0$, PreBitLT works exactly like BitLT: it computes $s_k = (1-a_k)d_k + \sum_{i=1}^{k-1}(1-a_i)(p_i - p_{i+1})$ and then $u_k = s_k \bmod 2$. For $u_\ell = (a'_\ell < b'_\ell)?\ 1 : 0$, $\ell \in [1, k-1]$, PreBitLT uses the intermediate sums $s_\ell = \sum_{i=1}^{\ell}(1-a_i)(p_i - p_{i+1})$. Since $s_\ell = \sum_{i=1}^{\ell}(1-a_i)d_i 2^{\sum_{j=i+1}^{k} d_j}$ and $p_{\ell+1} = 2^{\sum_{j=\ell+1}^{k} d_j}$, it follows that $s_\ell = p_{\ell+1}((1-a_\ell)d_\ell + \sum_{i=1}^{\ell-1}(1-a_i)d_i 2^{\sum_{j=i+1}^{\ell} d_j})$. Therefore, we can compute $u_\ell = (s_\ell / p_{\ell+1}) \bmod 2$, like in BitLT. Since $p_{\ell+1} \mid s_\ell$, the integer division can be efficiently computed in $\mathbb{Z}_q$, as $s_\ell(p_{\ell+1}^{-1} \bmod q)$. The multiplicative inverses $\{p_i^{-1}\}_{i=1}^k$ are efficiently computed by the protocol SufMulInv, a simple extension of the protocol SufMul (the additional computation does not need interaction)[2]. The protocol Mod2D is the variant of Mod2 with the input shared with polynomial of degree $2t$.

V. Conclusions

The protocols presented in this paper extend and improve the framework for secure computation with integer and fixed-point numbers introduced in [1] and [2].

We add as new building blocks the protocols PreMod2m, PreDiv2m and PreBitLT, efficient generalizations of the protocols Mod2m, Div2m and BitLT described in [1]. Using them, we obtain an efficient variant of a fundamental building block, BitDec, which computes the binary encoding of secret signed integers. This protocol fills a gap in the set of round-efficient primitives given in [1], by providing a variant of BitDec that runs in 3 rounds, instead of $\log(k)+1$ rounds, for k-bit inputs. Together, these building blocks have important applications in secure fixed-point and floating-point arithmetic.

Further efficiency gains are achieved by eliminating an interactive operation when secure multiplication of field elements is followed by additive hiding: the computation is completed in one round, instead of two, by locally randomizing the product shares (with PRZS [7]). This situation occurs very often. In particular, we use this technique to reduce the complexity of the typical secure fixed-point multiplication with truncation, from two on-line interaction rounds [2], to a single round.

Finally, we address the main performance issue of secure fixed-point arithmetic, by presenting a round-efficient secure division protocol. For usual bit-lengths, $k \in [64, 112]$, the protocol FXDiv computes $\tilde{a}/\tilde{b}$, with secret inputs and output, in 14 rounds (13 rounds if $\tilde{b}$'s sign is not secret). This is a substantial improvement with respect to the protocol described in [2], which needs 43 rounds (33 rounds if b's sign is not secret). This performance gain is achieved by combining the building blocks discussed above with an improved solution for computing the initial approximation.

The protocols have been implemented in Java, by extending our Java framework for secure multiparty computation (JSMC). Practical applicability, including accuracy, performance, and scalability, was demonstrated by solving linear programming problems with secret inputs and outputs [3]. We stress, however, that the optimizations related to joint generation of secret random values, pre-computation, share conversions and parallel computation have an important performance impact (see also [1], [2]). Without them, one should expect substantial performance degradation.

Secure floating-point computation has inherent performance issues, but also provides important benefits. Extending the framework in this direction is the target of on-going work. The goal is to offer both secure fixed-point and floating-point computation, so that applications can use the combination that better fits their requirements.

[2]The key issue in this generalization is the efficient computation of the set of prefix products. We evaluated several methods. The solution shown here, based on a clever remark in [9], minimizes the overall interaction complexity.

References

[1] O. Catrina, S. de Hoogh. "Improved Primitives for Secure Multiparty Integer Computation". In Security and Cryptography for Networks (SCN 2010), LNCS, vol. 6280, pp. 182-199, Springer, 2010.

[2] O. Catrina, A. Saxena. "Secure Computation with Fixed-point Numbers". In Financial Cryptography and Data Security (FC 2010), LNCS, vol. 6052, pp. 35-50, Springer, 2010.

[3] O. Catrina, S. de Hoogh. "Secure Multiparty Linear Programming Using Fixed-Point Arithmetic". In Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, LNCS, vol. 6345, pp. 134-150, Springer, 2010.

[4] F. Kerschbaum, A. Schropfer, A. Zilli, R. Pibernik, O. Catrina, S. de Hoogh, B. Schoenmakers, S. Cimato, E. Damiani. "Secure Collaborative Supply-Chain Management". IEEE Computer (Security and Privacy), vol. 44, no. 9, pg. 38-43, 2011.

[5] R. Cramer, I. Damgard, U. Maurer. "General Secure Multiparty Computation from any Linear Secret-Sharing Scheme". In EUROCRYPT 2000, LNCS, vol. 1807 , pp. 316-334. Springer, 2000.

[6] R. Cramer, I. Damgard, J. Nielsen. "Multiparty computation from threshold homomorphic encryption". In EUROCRYPT 2001, LNCS, vol. 2045, pp. 280-300. Springer, 2001.

[7] R. Cramer, I. Damgard, Y. Ishai. "Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation". In 2nd Theory of Cryptography Conference (TCC'05), pp. 342-362, 2005.

[8] I. Damgard, R. Thorbek. "Non-interactive Proofs for Integer Multiplication". In EUROCRYPT 2007, LNCS, vol. 4515, pp. 412-429. Springer, 2007.

[9] T. I. Reistad, T. Toft. "Linear, Constant-Rounds Bit-Decomposition". In International Conference on Information Security and Cryptology, LNCS, vol. 3329, pp. 245-257. Springer, 2009.

[10] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. "High-Performance Secure Multi-party Computation for Data Mining Applications". Int. J. Inf. Secur., 11(6):403-418, Nov. 2012.

[11] M. Aliasgari, M. Blanton, Y. Zhang, A. Steele. "Secure Computation on Floating Point Numbers". In 20th Annual Network and Distributed System Security Symposium (NDSS'13), 2013.

[12] L. Kamm, J. Willemson. "Secure Floating-point Arithmetic and Private Satellite Collision Analysis". Int. J. Inf. Secur., 14(6):531-548, Nov. 2015.

[13] M. Aliasgari, M. Blanton, F. Bayatbabolghani. "Secure Computation of Hidden Markov Models and Secure Floating-point Arithmetic in the Malicious Model". Int. J. Inf. Secur., 16(6):577-601, Nov 2017.

[14] P. Markstein. "Software Division and Square Root Using Goldschmidt's Algorithms". In 6th Conference on Real Numbers and Computers, pp. 146-157, 2004.

[15] M. D. Ercegovac, T. Lang. Digital Arithmetic. Morgan Kaufmann, 2003.