

Efficient Secure Floating-point Arithmetic using Shamir Secret Sharing

Octavian Catrina^a

University Politehnica of Bucharest, Bucharest, Romania

Keywords: Secure Multiparty Computation, Secure Floating-point Arithmetic, Secret Sharing.

Abstract: Successful deployment of privacy preserving collaborative applications, like statistical analysis, benchmarking, and optimizations, requires more efficient secure computation with real numbers. We present a complete family of protocols for secure floating-point arithmetic, constructed using a small set of building blocks that preserve data privacy using well known primitives based on Shamir secret sharing and related cryptographic techniques. Using new building blocks and optimizations and simpler secure fixed-point arithmetic, we obtain floating-point protocols with substantially improved efficiency.

1 INTRODUCTION

Secure computation enables groups of parties to run collaborative applications without having to reveal their private inputs: data privacy is preserved throughout the computation by cryptographic protocols. Various applications that require secure arithmetic with real numbers have been studied and implemented (Aliasgari et al., 2017; Bogdanov et al., 2018; Kamm and Willemson, 2015; Catrina and de Hoogh, 2010b). However, the performance penalty caused by cryptographic protocols remains an important deterrent for the deployment of these applications and motivates further research on improving current solutions (Dimitrov et al., 2016; Krips and Willemson, 2014).

Two frameworks based on secret sharing offer comprehensive support for multiparty secure computation with real numbers. The first framework (Catrina and de Hoogh, 2010a; Catrina and Saxena, 2010) provides a solid foundation for secure fixed-point computation, demonstrated by solving linear programming problems with private data (Catrina and de Hoogh, 2010b). Privacy is protected using well known primitives based on Shamir secret sharing and related techniques (Cramer et al., 2015; Cramer et al., 2005). Follow-up work added protocols for secure floating-point computation (Aliasgari et al., 2013) and related applications (Aliasgari et al., 2017). The other framework, Sharemind, was developed in parallel and relies on additive secret sharing. Its protocols for computing with real numbers have been gradually optimized (Krips and Willemson, 2014) and used in various applications (Bogdanov et al., 2018; Kamm and Willemson, 2015). These frameworks offer similar


security and performance for passive adversary (extension to active adversary is still expensive).

An initial goal of our project was to extend the first framework with building blocks and optimizations that offer better support for secure computation with real numbers. In this paper, we show how these extensions are used to obtain important performance gains for secure floating-point arithmetic.

The protocols provide the basic functionality and accuracy expected by typical applications, for practical range and precision settings. We focus on improving protocol performance and enabling trade-offs between performance and precision based on application requirements, rather than replicating the format and features specified in the IEEE Standard for Floating-Point Arithmetic (IEEE 754). Also, we aim at simplifying the protocols, by using a small set of components and constructions. We selected solutions that offer better trade-offs for the entire protocol family, rather than optimizing particular tasks. The paper is structured as follows. Section 2 is an overview of the secure computation framework, data encoding, and main building blocks. Section 3 presents the new family of protocols for secure floating-point arithmetic: addition and subtraction, multiplication, division, square root, and comparison. We summarize the main results in Section 4.

2 PRELIMINARIES

Secure Computation Model. The protocols presented in this paper use the secure computation framework described in (Catrina and de Hoogh, 2010a), which is based on standard primitives for secure com-

^a  <https://orcid.org/0000-0002-7498-9881>

putation using secret sharing (Cramer et al., 2015) and various optimizations presented in the literature (Cramer et al., 2005; Damgård et al., 2006; Damgård and Thorbek, 2007; Reistad and Toft, 2009). We start with an overview of this framework.

Suppose that $n > 2$ parties, P_1, P_2, \dots, P_n , communicate on secure channels and want to perform a joint 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 random share of each secret variable. Then, they compute with these shared variables to obtain the desired outputs, by running secure computation protocols.

Assuming perfectly secure channels and random number generators, these protocols offer perfect or statistical privacy: 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 $\Delta(X, Y) = \frac{1}{2} \sum_{v \in V \cup W} |Pr(X = v) - Pr(Y = v)|$. The distributions are perfectly indistinguishable if $\Delta(X, Y) = 0$ and statistically indistinguishable if $\Delta(X, Y)$ is negligible in some security parameter. With real-life secure channels and pseudo-random numbers, the protocols offer computational security.

The core primitives use Shamir secret sharing over a finite field \mathbb{F} . These primitives provide 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. Support for stronger adversary models can be added using various techniques, albeit with substantial performance degradation.

In this paper, we focus on protocols that 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} (Catrina and de Hoogh, 2010a; Catrina and Saxena, 2010). The parties locally compute addition/subtraction of shared field elements by adding/subtracting their own shares. Tasks that involve multiplication of shared values require interaction and are computed by dedicated protocols.

The protocols overcome the limitations of secure arithmetic with shared field elements, by combining secret sharing with additive or multiplicative hiding: for a shared variable $\llbracket x \rrbracket$ the parties jointly generate a secret random value $\llbracket r \rrbracket$, compute $\llbracket y \rrbracket = \llbracket x \rrbracket + \llbracket r \rrbracket$ or

Table 1: Complexity of core protocols (selection).

Protocol	Rounds	Int. Op.
$\llbracket a \rrbracket \leftarrow \text{Share}(a)$	1	1
$a \leftarrow \text{Reveal}(\llbracket a \rrbracket)$	1	1
$\llbracket c \rrbracket \leftarrow \llbracket a \rrbracket + \llbracket b \rrbracket$	0	0
$\llbracket c \rrbracket \leftarrow a + \llbracket b \rrbracket$	0	0
$\llbracket c \rrbracket \leftarrow a \llbracket b \rrbracket$	0	0
$\llbracket c \rrbracket \leftarrow \llbracket a \rrbracket \llbracket b \rrbracket$	1	1

$\llbracket y \rrbracket = \llbracket x \rrbracket \cdot \llbracket r \rrbracket$ and reveal y ; this is 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 $\Delta(x + r \bmod q, r) = 0$ and $\Delta(xr \bmod q, r) = 0$, hence perfect privacy. 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 substantially simplify the protocols by avoiding wraparound modulo q , although they require larger q for a given data range.

We evaluate the protocols using complexity metrics that focus on interaction between parties. Communication complexity measures the amount of data sent by each party. For our protocols, a suitable metric is the number of 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 and is relevant for network latency. Table 1 shows the complexity of the core primitives.

The protocols offer best performance for implementations that apply the following basic optimizations. 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) (Cramer et al., 2005) and its integer variant (RISS) (Damgård and Thorbek, 2007) to generate without interaction shared random field elements and integers, and random sharings of 0. Some shared random values cannot be generated without interaction (e.g., random bits shared in \mathbb{Z}_q). We indicate separately the communication complexity of the pre-computation round.

Data Types and Data Encoding. We consider secure computation with the following data types: binary values, signed integers, fixed-point numbers, and floating-point numbers. For secure computation, they are encoded in a finite field \mathbb{F} . We distinguish different representations of a number as follows: we denote \tilde{x} a fixed-point number, \bar{x} the integer value encoding \tilde{x} , x the field element that encodes \tilde{x} , and $\llbracket x \rrbracket$ a sharing of x ; a floating-point number is denoted \hat{x} . The no-

tation $x = (\text{condition})? a : b$ means that x is assigned the value a when $\text{condition} = \text{true}$ and b otherwise.

Logical values $\text{false}, \text{true}$ 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} (Catrina and de Hoogh, 2010a). We denote $\llbracket a \rrbracket \wedge \llbracket b \rrbracket = \llbracket a \rrbracket \llbracket b \rrbracket = \llbracket a \wedge b \rrbracket$ (AND), $\llbracket a \rrbracket \vee \llbracket b \rrbracket = \llbracket a \rrbracket + \llbracket b \rrbracket - \llbracket a \rrbracket \llbracket b \rrbracket = \llbracket a \vee b \rrbracket$ (OR) and $\llbracket a \rrbracket \oplus \llbracket b \rrbracket = \llbracket a \rrbracket + \llbracket b \rrbracket - 2\llbracket a \rrbracket \llbracket b \rrbracket = \llbracket a \oplus b \rrbracket$ (XOR).

Signed integer types are defined as $\mathbb{Z}_{(k)} = \{\bar{x} \in \mathbb{Z} \mid \bar{x} \in [-2^{k-1}, 2^{k-1} - 1]\}$. They are encoded in \mathbb{Z}_q by the function $\text{fld} : \mathbb{Z}_{(k)} \mapsto \mathbb{Z}_q$, $\text{fld}(\bar{x}) = \bar{x} \bmod q$, for a prime $q > 2^{k+\kappa}$, where κ is the security parameter (similar to two's complement encoding). This method enables efficient secure integer arithmetic using secure arithmetic in \mathbb{Z}_q : for any $\bar{x}_1, \bar{x}_2 \in \mathbb{Z}_{(k)}$ and $\odot \in \{+, -, \cdot\}$, we have $\bar{x}_1 \odot \bar{x}_2 = \text{fld}^{-1}(\text{fld}(\bar{x}_1) \odot \text{fld}(\bar{x}_2))$; also, if $\bar{x}_2 \mid \bar{x}_1$ then $\bar{x}_1/\bar{x}_2 = \text{fld}^{-1}(\text{fld}(\bar{x}_1) \cdot \text{fld}(\bar{x}_2)^{-1})$.

Signed fixed-point types are sets of rational numbers defined as $\mathbb{Q}_{(k,f)}^{FX} = \{\bar{x} \in \mathbb{Q} \mid \bar{x} = \bar{x}2^{-f}, \bar{x} \in \mathbb{Z}_{(k)}\}$, for $f < k$. They are obtained by sampling at 2^{-f} intervals the range of real numbers $[-2^{k-f-1}, 2^{k-f-1} - 2^{-f}]$. The value 2^{-f} is the resolution of the fixed-point type. $\mathbb{Q}_{(k,f)}^{FX}$ is mapped to $\mathbb{Z}_{(k)}$ by the function $\text{int} : \mathbb{Q}_{(k,f)}^{FX} \mapsto \mathbb{Z}_{(k)}$, $\bar{x} = \text{int}_f(\bar{x}) = \bar{x}2^f$ and encoded in \mathbb{Z}_q as described above. Secure multiplication and division of fixed-point numbers require $q > 2^{2k+\kappa}$.

Floating-point numbers $\hat{x} \in \mathbb{Q}_{(l,g)}^{FL}$ are 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, $\bar{p} \in \mathbb{Z}_{(g)}$ is the signed exponent, $s = (\hat{x} < 0)? 1 : 0$, and $z = (\hat{x} = 0)? 1 : 0$. The value of the number is $\hat{x} = (1 - 2s) \cdot \bar{v} \cdot 2^{\bar{p}}$. If $\hat{x} = 0$ then $z = 1$, $\bar{v} = 0$, and $\bar{p} = -2^{g-1}$. This encoding of $\hat{x} = 0$ simplifies secure addition with minimal negative effects on other operations. The integer significand and exponent are encoded as described above.

The parameters k, f, ℓ and g are not secret. The protocols work for any setting of these parameters that satisfies the type definitions. The applications usually need $k \in [32, 128]$, $k = 2f$, $\ell \in [24, 64]$ and $g \in [8, 15]$, depending on range and accuracy requirements.

The floating-point protocols are constructed using a subset of the building blocks introduced in (Catrina and de Hoogh, 2010a; Catrina and Saxena, 2010), enhanced by optimizations added in (Catrina, 2018). All building blocks rely on the secure computation model described above for their own security and secure composition. We summarize in the following their functionality and the optimizations¹. Table 2 lists their online and precomputation complexity.

¹Further details are available in the Appendix.

Multiplication and Inner Product. Standard multiplication of Shamir-shared field elements requires an interaction. However, this interaction can be avoided in some cases. Denote $\llbracket x \rrbracket_{u,i}$ the share of x owned by P_i , for a random polynomial of degree u ; the default value of the degree is t . The multiplication protocol computes $\llbracket c \rrbracket \leftarrow \llbracket a \rrbracket \llbracket b \rrbracket$ as follows (Cramer et al., 2015): for all $i \in [1, n]$, P_i locally computes $\llbracket a \rrbracket_i \llbracket b \rrbracket_i$; the result is $\llbracket c \rrbracket_{2t,i}$, a share of c for a non-random polynomial of degree $2t$ (product of the polynomials used to share a and b); then, P_i shares the value $\llbracket c \rrbracket_{2t,i}$ by sending to the others $\llbracket \llbracket c \rrbracket_{2t,i} \rrbracket_j$, for $j \in [1, n]$, $j \neq i$; finally, P_i computes its own share $\llbracket c \rrbracket_i$ from the received shares, by Lagrange interpolation.

A first optimization applies to multiplications followed by additive hiding: $d \leftarrow \text{Reveal}(\llbracket a \rrbracket \llbracket b \rrbracket + \llbracket r \rrbracket)$. With standard protocols, this computation needs 2 rounds. We can avoid the first round by locally randomizing the share products: for all $i \in [1, n]$, party i computes $\llbracket c \rrbracket_{2t,i} \leftarrow \llbracket a \rrbracket_i \llbracket b \rrbracket_i + \llbracket 0 \rrbracket_{2t,i}$, where $\llbracket 0 \rrbracket_{2t,i}$ are pseudo-random shares of 0 generated with PRZS($2t$) (Cramer et al., 2005). We denote $\llbracket a \rrbracket * \llbracket b \rrbracket$ this local operation. The computation can now be completed with a single interaction: $d \leftarrow \text{RevealD}(\llbracket a \rrbracket * \llbracket b \rrbracket + \llbracket r \rrbracket)$, where RevealD is the secret reconstruction protocol for polynomials of degree $2t$.

This situation occurs very often. In particular, many of the protocols discussed below use 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'. The difference is that they use RevealD instead of Reveal .

Another optimization is used to compute the inner product of two vectors, $\llbracket c \rrbracket \leftarrow \sum_{k=1}^m \llbracket a_k \rrbracket \llbracket b_k \rrbracket$, in the protocol InnerProd : the parties locally compute the inner product of their own shares and re-share the result. Thus, InnerProd needs a single interaction (instead of m interactions). If InnerProd is followed by additive hiding of its output, we can also use the previous optimization. We call InnerProdD a variant that locally computes $\llbracket c \rrbracket_{2t,i} \leftarrow \sum_{k=1}^m \llbracket a_k \rrbracket_i \llbracket b_k \rrbracket_i + \llbracket 0 \rrbracket_{2t,i}$.

Multiplication and Division by 2^m . The floating-point arithmetic protocols are built using a small set of related protocols that efficiently compute $\bar{b} = \bar{a} \cdot 2^m$ and $\bar{c} \approx \bar{a}/2^m$, for secret $\bar{a}, \bar{b}, \bar{c} \in \mathbb{Z}_{(k)}$ and public or secret integer $m \in [0, k-1]$.

If m is public we compute $\bar{a} \cdot 2^m$ without interaction. To compute $\bar{a}/2^m$, we use the protocols Div2m and Div2mP , introduced in (Catrina and de Hoogh, 2010a; Catrina and Saxena, 2010). Div2m rounds to $-\infty$ and Div2mP rounds probabilistically to the nearest integer. We denote their outputs $\lfloor \bar{a}/2^m \rfloor$ and $\lfloor \bar{a}/2^m \rfloor$, respectively. Div2mP computes $\bar{c} =$

Table 2: Complexity of the main building blocks used in this paper, for inputs $\bar{a}, \bar{x} \in \mathbb{Z}_{(k)}$.

Protocol	Task	Rounds	Inter. op.	Precomp.
Div2m($\llbracket a \rrbracket, k, m$)	$\lfloor \bar{a}/2^m \rfloor$	3	$m + 2$	$3m$
Div2mP($\llbracket a \rrbracket, k, m$)	$\lceil \bar{a}/2^m \rceil$	1	1	m
Div2($\llbracket a \rrbracket, k$)	$\lfloor \bar{a}/2 \rfloor$	1	1	1
LTZ($\llbracket a \rrbracket, k$)	$(\bar{a} < 0)? 1 : 0$	3	$k + 1$	$3k$
EQZ($\llbracket a \rrbracket, k$)	$(\bar{a} = 0)? 1 : 0$	3	$\log k + 2$	$k + 3 \log k$
SufOr($\{\llbracket a_i \rrbracket\}_{i=1}^k$)	$\{\bigvee_{j=i}^k a_j\}_{i=1}^k$	2	$2k - 1$	$3k$
SufMul($\{\llbracket a_i \rrbracket\}_{i=1}^k$)	$\{\prod_{j=i}^k a_j\}_{i=1}^k$	1	k	$2k - 1$
PreDiv2m($\llbracket a \rrbracket, k, m$)	$\{\lfloor \bar{a}/2^i \rfloor\}_{i=1}^m$	3	$2m + 1$	$4m$
PreDiv2mP($\llbracket a \rrbracket, k, m$)	$\{\lceil \bar{a}/2^i \rceil\}_{i=1}^m$	1	1	m
Int2MaskG($\llbracket x \rrbracket, k, m$)	$\{(\bar{x} = i)? 1 : 0\}_{i=0}^{k-1}$	5	$k + m + 2$	$2k + 6m$

$\lfloor \bar{a}/2^m \rfloor + u$, where $u = 1$ 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 protocols, the rounding error is $|\delta| < 1$ and the output is exact if 2^m divides \bar{a} . Div2mP is much more efficient (Table 2) and its output is likely more accurate. Div2 is a more efficient solution for $\lfloor \bar{a}/2 \rfloor$. Finally, the comparison protocol LTZ uses Div2m to compute $s = (\bar{a} < 0)? 1 : 0 = -\lfloor \bar{a}/2^{k-1} \rfloor$.

If m is secret we have to extend the collection of building blocks in (Catrina and de Hoogh, 2010a). The goal is to use the following constructions. We start by computing the secret bits $\{x_i\}_{i=0}^{k-1}$, $x_i = (m = i)? 1 : 0$. This allows us to locally compute $2^m = \sum_{i=0}^{k-1} x_i 2^i$ and then $\bar{a} \cdot 2^m$. Moreover, we can use a similar method for $\bar{a}/2^m$: compute the secret integers $\bar{d}_i = \lfloor \bar{a}/2^i \rfloor$ and the inner product $\bar{a}/2^m = \sum_{i=0}^{k-1} x_i \bar{d}_i$.

The protocol PreDiv2m, suggested in (Catrina, 2018), is a generalization of Div2m that efficiently computes $\{\lfloor \bar{a}/2^i \rfloor\}_{i=1}^m$ with secret inputs and outputs. Surprisingly, it performs a much more complex task than Div2m in the same number of rounds, with a modest increase of the communication complexity (Table 2). PreDiv2mP is a generalization of Div2mP that computes $\{\lceil \bar{a}/2^i \rceil\}_{i=1}^m$ with probabilistic rounding to nearest, with the same complexity as Div2mP.

Protocol 1, Int2MaskG, is a generic construction for computing $\{x_i\}_{i=0}^{k-1}$, $x_i = (\bar{x} = i)? 1 : 0$, using Lagrange polynomial interpolation in \mathbb{Z}_q , $q > k$, adapted to our tasks. Given a secret $\bar{x} \in [0, 2^{m-1} - 1]$ and public $k \leq 2^{m-1}$, it returns the secret bits $\{x_i\}_{i=0}^{k-1}$ such that $x_i = 1$ if $\bar{x} < k$ and $i = \bar{x}$, otherwise $x_i = 0$.

Steps 1-2 map \bar{x} to $\bar{x}' = (\bar{x} < k)? \bar{x} : k$, $\bar{x}' \in [0, k]$. Let $\alpha = \bar{x}' + 1$. We compute $\{x_i\}_{i=0}^{k-1}$ by evaluating the functions $f_i : [1, k + 1] \rightarrow \{0, 1\}$, $f_i(\alpha) = (\alpha = i + 1)? 1 : 0$, for $i \in [0, k - 1]$, using their interpolation polynomials $f_i(\alpha) = \sum_{j=0}^k a_{i,j} \alpha^j$. The coefficients $a_{i,j}$ are pre-computed from public information (the points

that define $\{f_i\}_{i=0}^{k-1}$). Steps 3-4 compute $\{\alpha^i\}_{i=1}^k$ using PreMul and then $x_i = f_i(\alpha)$, for $i \in [0, k - 1]$ (we set $\alpha = \bar{x}' + 1$ because PreMul requires non-zero inputs). The online complexity is 5 rounds and $k + m + 2$ interactive operations.

P 1: Int2MaskG($\llbracket x \rrbracket, k, m$).

- 1 $\llbracket d \rrbracket \leftarrow \text{LTZ}(\llbracket x \rrbracket - k, m + 1)$;
 - 2 $\llbracket x' \rrbracket \leftarrow \llbracket d \rrbracket \llbracket x \rrbracket + (1 - \llbracket d \rrbracket)k$;
 - 3 $\{\llbracket y_j \rrbracket\}_{j=1}^k \leftarrow \text{PreMul}(\{\llbracket x' \rrbracket + 1\}_{i=1}^k)$;
 - 4 foreach $i \in [0, k - 1]$ do
 - 5 $\llbracket x_i \rrbracket \leftarrow a_{i,0} + \sum_{j=1}^k a_{i,j} \llbracket y_j \rrbracket$;
 - 6 return $\{\llbracket x_i \rrbracket\}_{i=0}^{k-1}$;
-

3 SECURE FLOATING-POINT ARITHMETIC

We present a family of floating-point arithmetic protocols for addition, subtraction, multiplication, division, square root, and comparison. We focus on solutions that offer the best tradeoffs for the entire family and a broader range of applications. All protocols are constructed using the techniques discussed in Section 2, that support secure protocol composition. The same security arguments apply to the entire family, so we do not repeat them for each protocol.

Converting Fixed-point Numbers to Floating-point Numbers. Given a fixed-point number $\bar{a} \in \mathbb{Q}_{(k,f)}^{FX}$, Protocol 2, FX2FL, computes $\langle \bar{v}, \bar{p}, s, z \rangle$ so that $\hat{a} = (1 - 2s)\bar{v}2^{\bar{p}} \in \mathbb{Q}_{(\ell,g)}^{FL}$, $\hat{a} \approx \bar{a}$ and $z = (\hat{a} = 0)? 1 : 0$, with secret input and output. In particular, for $f = 0$, the input is an integer $\bar{a} \in \mathbb{Z}_{(k)}$ and the output is $\hat{a} \in \mathbb{Q}_{(\ell,g)}^{FL}$ so that $\hat{a} \approx \bar{a}$. FX2FL is also used for normalizing the output of floating-point arithmetic protocols.

The computation can be summarized as follows. Let $\bar{a} = \bar{a}2^f$. Recall that $\bar{a} \in [-(2^{k-1} - 1), 2^{k-1} - 1]$ and $\bar{v} \in [2^{\ell-1}, 2^\ell - 1] \cup \{0\}$. If $\bar{a} = 0$ we set $\bar{v} = 0$ and $\bar{p} = -2^{s-1}$. Otherwise, $|\bar{a}| \in [2^{m-1}, 2^m - 1]$ for some secret $m \in [1, k-1]$ and we have to compute $\bar{v} = |\bar{a}|2^{\ell-m}$ and $\bar{p} = -f - \ell + m$. When $k-1 > \ell$ we have 2 cases: if $m \leq \ell$ then $\bar{v} = |\bar{a}|2^{\ell-m}$; if $m > \ell$ then $\bar{v} = \lfloor |\bar{a}|/2^{-\ell+m} \rfloor$. If $k-1 \leq \ell$ then $m \leq \ell$ and hence $\bar{v} = |\bar{a}|2^{\ell-m}$. Therefore, if $m \leq \ell$ the output is $\hat{a} = \bar{a}$, otherwise $\hat{a} \approx \bar{a}$, with relative error $\varepsilon < 2^{-\ell}$, due to the truncation of \bar{a} .

Steps 1-6 compute $s = (\bar{a} < 0)? 1 : 0$ and $z = (\bar{a} = 0)? 1 : 0$, together with data used in steps 6-10 for computing \bar{v} and \bar{p} : $\{\bar{b}_i\}_{i=0}^{k-2} = \{\lfloor |\bar{a}|/2^i \rfloor\}_{i=0}^{k-2}$, $\{a_i\}_{i=0}^{k-2}$, the binary encoding of $|\bar{a}|$, and $\{c_i\}_{i=0}^{k-2} = \{\bigvee_{j=i}^{k-2} a_j\}_{i=0}^{k-2}$. Note that $\bar{b}_0 = \bar{a}(1-2s) = |\bar{a}|$ and $c_0 = \bigvee_{j=0}^{k-2} a_j = 1 - z$. By using PreDiv2mD instead of PreDiv2m, the multiplication $\llbracket a \rrbracket(1 - 2\llbracket s \rrbracket)$ is computed without interaction, saving one round.

P 2: FX2FL($\llbracket a \rrbracket, k, f, \ell, g$).

```

1  $\llbracket s \rrbracket \leftarrow \text{LTZ}(\llbracket a \rrbracket, k); \llbracket s' \rrbracket \leftarrow 1 - 2\llbracket s \rrbracket;$ 
2  $\{\llbracket b_i \rrbracket\}_{i=0}^{k-2} \leftarrow \text{PreDiv2mD}(\llbracket a \rrbracket * \llbracket s' \rrbracket, k, k-2);$ 
3  $\llbracket a_{k-2} \rrbracket \leftarrow \llbracket b_{k-2} \rrbracket;$ 
4 foreach  $i \in [0, k-3]$  do  $\llbracket a_i \rrbracket \leftarrow \llbracket b_i \rrbracket - 2\llbracket b_{i+1} \rrbracket;$ 
5  $\{\llbracket c_i \rrbracket\}_{i=0}^{k-2} \leftarrow \text{SufOr}(\{\llbracket a_i \rrbracket\}_{i=0}^{k-2});$ 
6  $\llbracket z \rrbracket \leftarrow 1 - \llbracket c_0 \rrbracket;$ 
7 foreach  $i \in [0, k-3]$  do  $\llbracket d_i \rrbracket \leftarrow \llbracket c_i \rrbracket - \llbracket c_{i+1} \rrbracket;$ 
8  $\llbracket d_{k-2} \rrbracket \leftarrow \llbracket c_{k-2} \rrbracket;$ 
9 if  $k-1 > \ell$  then  $\llbracket v \rrbracket \leftarrow \llbracket b_0 \rrbracket \sum_{i=0}^{\ell-1} 2^{\ell-i-1} \llbracket d_i \rrbracket +$ 
    $\sum_{i=0}^{k-\ell-2} \llbracket d_{\ell+i} \rrbracket \llbracket b_{i+1} \rrbracket;$ 
10 else  $\llbracket v \rrbracket \leftarrow 2^{\ell-k+1} \llbracket b_0 \rrbracket \sum_{i=0}^{k-2} 2^{k-i-2} \llbracket d_i \rrbracket;$ 
11  $\llbracket p \rrbracket \leftarrow (-f - \ell)(1 - \llbracket z \rrbracket) + \sum_{i=0}^{k-2} \llbracket c_i \rrbracket - \llbracket z \rrbracket 2^{s-1};$ 
12 return  $(\llbracket v \rrbracket, \llbracket p \rrbracket, \llbracket s \rrbracket, \llbracket z \rrbracket);$ 

```

Steps 7-10 compute \bar{v} . We start by computing $d_i = (i = m-1)? 1 : 0$, for $i \in [0, k-2]$. If $k-1 > \ell$ we have to compute $\bar{v}_1 = |\bar{a}|2^{\ell-m}$, if $m \in [1, \ell]$, and $\bar{v}_2 = \lfloor |\bar{a}|/2^{-\ell+m} \rfloor$, if $m \in [\ell+1, k-1]$. At least one of these values is 0, so step 9 obliviously handles both cases by computing $\bar{v}_1 = \bar{b}_0 \sum_{i=0}^{\ell-1} 2^{\ell-i-1} d_i = |\bar{a}|2^{\ell-m}$, $\bar{v}_2 = \sum_{i=0}^{k-\ell-2} d_{\ell+i} \bar{b}_{i+1} = \bar{b}_{m-\ell} = \lfloor |\bar{a}|/2^{-\ell+m} \rfloor$ and $\bar{v} = \bar{v}_1 + \bar{v}_2$. If $k \leq \ell+1$ then $m \leq \ell$ and $\bar{v} = |\bar{a}|2^{\ell-m}$. This case is computed in step 10: $\bar{v} = 2^{\ell-k+1} \bar{b}_0 \sum_{i=0}^{k-2} 2^{k-i-2} d_i = 2^{\ell-k+1} |\bar{a}| 2^{k-m-1} = |\bar{a}| 2^{\ell-m}$.

Step 11 computes \bar{p} . If $\bar{a} \neq 0$ then $\sum_{i=0}^{k-2} c_i = m$, since $c_i = 1$ for $i \in [0, m-1]$ and $c_i = 0$ for $i \in [m, k-2]$; otherwise, $\sum_{i=0}^{k-2} c_i = 0$. Therefore, if $\bar{a} \neq 0$ then $z = 0$ and $\bar{p} = -f - \ell + m$, otherwise $z = 1$ and $\bar{p} = -2^{s-1}$, as required.

The online complexity is 9 rounds and $5k+3$ interactive operations. If the sign is not secret, we can

skip step 1 and the complexity becomes 6 rounds and $4k+2$ operations. FX2FL is simpler and more efficient than the protocol given in (Aliasgari et al., 2013), which needs $\log k + 12$ rounds and more than $(\log k + 3)k$ operations. The improvement is due to more efficient solutions enabled by PreDiv2m for computing the secret index m and the multiplication and division by secret $2^{\ell-m}$.

We also need FX2FLE, a general tool for normalizing the output of floating-point arithmetic protocols. FX2FLE is a variant of FX2FL that takes a secret integer \bar{x} as additional input and returns $\langle \bar{v}, \bar{p}, s, z \rangle$ so that $\hat{a} = (1-2s)\bar{v}2^{\bar{p}} \in \mathbb{Q}_{(\ell, g)}^{FL}$ and $\hat{a} \approx 2^{\bar{x}} \bar{a}$. The difference is that step 11 computes $\llbracket p \rrbracket \leftarrow (\llbracket x \rrbracket + \llbracket m \rrbracket) - f - \ell)(1 - \llbracket z \rrbracket) - \llbracket z \rrbracket 2^{s-1}$, in parallel with the computation of \bar{v} (the round complexity is the same).

Floating-point Addition and Subtraction. Protocol 3, AddFL, computes $\hat{a} = \hat{a}_1 + \hat{a}_2$, for secret $\hat{a}_1, \hat{a}_2, \hat{a} \in \mathbb{Q}_{(\ell, g)}^{FL}$, $\hat{a}_1 = (1-2s_1)\bar{v}_1 2^{\bar{p}_1}$, $\hat{a}_2 = (1-2s_2)\bar{v}_2 2^{\bar{p}_2}$, and $\hat{a} = (1-2s)\bar{v} 2^{\bar{p}}$. AddFL can also compute $\hat{a} = \hat{a}_1 - \hat{a}_2$ by setting $s_2 = 1 - s_2$.

The basic idea is to align the inputs' radix point, add the significands and normalize the result using FX2FLE. To simplify the notation, suppose $\hat{a}_1 \geq 0$, $\hat{a}_2 \geq 0$, and $\bar{p}_1 \geq \bar{p}_2$. We want \bar{v} and \bar{p} so that $\bar{v} 2^{\bar{p}} \approx \bar{v}_1 2^{\bar{p}_1} + \bar{v}_2 2^{\bar{p}_2}$. We can align to the larger exponent, by setting $\bar{p} = \bar{p}_1$ and $\bar{v} = \bar{v}_1 + \lfloor \bar{v}_2/2^{\bar{p}_1-\bar{p}_2} \rfloor$, or to the smaller exponent, by setting $\bar{p} = \bar{p}_2$ and $\bar{v} = \bar{v}_1 2^{\bar{p}_1-\bar{p}_2} + \bar{v}_2$. Which method is better? Multiplication by secret $2^{\bar{p}_1-\bar{p}_2}$ is simpler than division, but here it is inefficient, since the result can be huge. One solution is to combine the methods: use the first method when $\bar{p}_1 - \bar{p}_2 \geq \ell$, because $\lfloor \bar{v}_2/2^{\bar{p}_1-\bar{p}_2} \rfloor = 0$; otherwise, use the second method, because $\bar{v}_1 2^{\bar{p}_1-\bar{p}_2} < 2^{2\ell}$. AddFL uses the first method, which can be implemented more efficiently with the new building blocks.

Steps 1-3 swap the inputs if $\bar{p}_1 < \bar{p}_2$: $(\bar{v}'_1, \bar{p}'_1) = (\bar{p}_1 < \bar{p}_2)? ((1-2s_2)\bar{v}_2, \bar{p}_2) : ((1-2s_1)\bar{v}_1, \bar{p}_1)$ and $(\bar{v}'_2, \bar{p}'_2) = (\bar{p}_1 < \bar{p}_2)? ((1-2s_1)\bar{v}_1, \bar{p}_1) : ((1-2s_2)\bar{v}_2, \bar{p}_2)$. Swap computes $(c = 1)? (y, x) : (x, y)$ with secret inputs and outputs. Since we encode $\hat{a} = 0$ as $\bar{v} = 0$ and $\bar{p} = -2^{s-1}$ (smallest value), null operands are not special cases: if $\hat{a}_2 = 0$ then $\bar{p}_1 \geq \bar{p}_2$ and $\bar{v}_2 = 0$, so the protocol sets $\bar{p} = \bar{p}_1$ and $\bar{v} = \bar{v}_1$; if $\hat{a}_1 = 0$, $\bar{p}_1 < \bar{p}_2$ and the operands are swapped.

Let $\Delta = \bar{p}'_1 - \bar{p}'_2 \geq 0$. Steps 4-6 compute $\{x_i\}_{i=0}^{\ell-1} = \{(\Delta = i)? 1 : 0\}_{i=0}^{\ell-1}$ and $\{\bar{d}_i\}_{i=0}^{\ell-1} = \{\lfloor \bar{v}'_2/2^i \rfloor\}_{i=0}^{\ell-1}$ (in parallel), then step 7 computes $\bar{v}'_3 = \bar{v}'_1 + \sum_{i=0}^{\ell-1} x_i \bar{d}_i = \bar{v}'_1 + \lfloor \bar{v}'_2/2^\Delta \rfloor$. Step 8 normalizes the result. If $s_1 \neq s_2$ and $\Delta = 1$, $|\bar{v}_1 - \bar{v}_2/2|$ can be close to the rounding error, compromising the accuracy. This is avoided by setting $\bar{v}'_1 = 2\bar{v}_1$, $\bar{v}'_2 = 2\bar{v}_2$, so that the division is exact,

and invoking FX2FLE with $k = \ell + 3$ and $\bar{p}'_1 = \bar{p}'_1 - 1$.

P 3: AddFL($\{\llbracket v_i \rrbracket, \llbracket p_i \rrbracket, \llbracket s_i \rrbracket\}_{i=1}^2$).

- 1 $\llbracket c \rrbracket \leftarrow \text{LTZ}(\llbracket p_1 \rrbracket - \llbracket p_2 \rrbracket, g + 1)$;
- 2 $\llbracket v_1 \rrbracket \leftarrow \llbracket v_1 \rrbracket(1 - 2\llbracket s_1 \rrbracket)$; $\llbracket v_2 \rrbracket \leftarrow \llbracket v_2 \rrbracket(1 - 2\llbracket s_2 \rrbracket)$;
- 3 $\{\llbracket v'_i \rrbracket, \llbracket p'_i \rrbracket\}_{i=1}^2 \leftarrow \text{Swap}(\llbracket c_0 \rrbracket, \{\llbracket v_i \rrbracket, \llbracket p_i \rrbracket\}_{i=1}^2)$;
- 4 $\{\llbracket x_i \rrbracket\}_{i=0}^{\ell-1} \leftarrow \text{Int2MaskG}(\llbracket p'_1 \rrbracket - \llbracket p'_2 \rrbracket, \ell, g + 1)$;
- 5 $\llbracket d_0 \rrbracket \leftarrow \llbracket v'_2 \rrbracket$;
- 6 $\{\llbracket d_i \rrbracket\}_{i=1}^{\ell-1} \leftarrow \text{PreDiv2mP}(\llbracket v'_2 \rrbracket, \ell + 1, \ell - 1)$;
- 7 $\llbracket v'_3 \rrbracket \leftarrow \llbracket v'_1 \rrbracket + \sum_{i=0}^{\ell-1} \llbracket x_i \rrbracket \llbracket d_i \rrbracket$;
- 8 $(\llbracket v \rrbracket, \llbracket p \rrbracket, \llbracket s \rrbracket, \llbracket z \rrbracket) \leftarrow \text{FX2FLE}(\llbracket v'_3 \rrbracket, \llbracket p'_1 \rrbracket, \ell + 2, 0, \ell, g)$;
- 9 **return** $(\llbracket v \rrbracket, \llbracket p \rrbracket, \llbracket s \rrbracket, \llbracket z \rrbracket)$;

The online complexity AddFL is 19 rounds and $6\ell + 2g + 26$ interactive operations. For operands with the same, known sign the complexity is 16 rounds and $5\ell + 2g + 10$ operations (due to simpler normalization). With minor changes, AddFL also works when one operand is public, with roughly the same complexity. The protocol proposed in (Aliasgari et al., 2013) needs $\log \ell + 30$ rounds and more than $(\log \ell + 14)\ell + 9g$ operations. The improvement is due to more efficient building blocks and the simpler algorithm enabled by PreDiv2mP and Int2MaskG.

Floating-point multiplication. Protocol 4, MulFL, computes $\hat{a} \approx \hat{a}_1 \hat{a}_2$, for secret $\hat{a}_1, \hat{a}_2, \hat{a} \in \mathbb{Q}_{(\ell, g)}^{FL}$, $\hat{a}_1 = (1 - 2s_1)\bar{v}_1 2^{\bar{p}_1}$, $\hat{a}_2 = (1 - 2s_2)\bar{v}_2 2^{\bar{p}_2}$, $\hat{a} = (1 - 2s)\bar{v} 2^{\bar{p}}$. The protocol computes $\bar{v}_3 = \bar{v}_1 \bar{v}_2$ and $\bar{p}_3 = \bar{p}_1 + \bar{p}_2$ and normalizes the result. Since $\bar{v}_3 \in [2^{2\ell-2}, 2^{2\ell} - 2^{\ell+1} + 1] \cup \{0\}$, normalization is easy: 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 $\bar{p} = \bar{p}_3 + \ell$. Also, we efficiently compute $s = s_1 \oplus s_2$, $z = z_1 \vee z_2$, and $\bar{p} = \bar{p}(1 - z) - z 2^{g-1}$.

We can reduce the communication complexity by modifying the algorithm as follows: compute $\bar{v}_3 = \lfloor \bar{v}_1 \bar{v}_2 / 2^{\ell-1} \rfloor \in [2^{\ell-1}, 2^{\ell+1} - 2] \cup \{0\}$ and $\bar{p}_3 = \bar{p}_1 + \bar{p}_2 + \ell - 1$; if $\bar{v}_3 < 2^\ell$ then set $\bar{v} = \bar{v}_3$ and $\bar{p} = \bar{p}_3$; otherwise, set $\bar{v} = \lfloor \bar{v}_3 / 2 \rfloor$ and $\bar{p} = \bar{p}_3 + 1$. A similar method is used in (Aliasgari et al., 2013). Our protocol is an optimized variant.

Steps 1-3 compute $(\bar{v}_3, \bar{p}_3, s, z)$ as explained above and step 4 normalizes the result using Protocol 5, NormFLS. We use fast truncation with Div2mP, because \bar{v}_3 is in the range $[2^{\ell-1}, 2^{\ell+1} - 1] \cup \{0\}$ required for simple normalization regardless of the rounding method (actually, we use Div2mPD, so that the multiplication $\llbracket v_1 \rrbracket \llbracket v_2 \rrbracket$ is computed without interaction).

NormFLS normalizes $\bar{v} \in [2^{\ell-1}, 2^{\ell+1} - 1] \cup \{0\}$ using the algorithm described above: it computes (in parallel) $b = (\bar{v} < 2^\ell)$? $1 : 0$ and $\bar{v}'' = \lfloor \bar{v} / 2 \rfloor$, then $\bar{v}' = (b = 1) ? \bar{v} : \bar{v}''$ and $\bar{p}' = (b = 1) ? \bar{p}(1 - z) :$

$(\bar{p} + 1)(1 - z)$, $\bar{p}' = \bar{p}' - z 2^{g-1}$.

The online complexity of MulFL is 5 rounds and $\ell + 9$ interactive operations (instead of 11 rounds and $8\ell + 10$ operations reported in (Aliasgari et al., 2013)). MulFL also works when one of the operands is public, with minor changes and slightly lower complexity.

P 4: MulFL($\{\llbracket v_i \rrbracket, \llbracket p_i \rrbracket, \llbracket s_i \rrbracket, \llbracket z_i \rrbracket\}_{i=1}^2$).

- 1 $\llbracket v_3 \rrbracket \leftarrow \text{Div2mPD}(\llbracket v_1 \rrbracket * \llbracket v_2 \rrbracket, 2\ell, \ell - 1)$;
- 2 $\llbracket s \rrbracket \leftarrow \llbracket s_1 \rrbracket \oplus \llbracket s_2 \rrbracket$; $\llbracket z \rrbracket \leftarrow \llbracket z_1 \rrbracket \vee \llbracket z_2 \rrbracket$;
- 3 $\llbracket p_3 \rrbracket \leftarrow \llbracket p_1 \rrbracket + \llbracket p_2 \rrbracket + \ell - 1$;
- 4 $(\llbracket v \rrbracket, \llbracket p \rrbracket) \leftarrow \text{NormFLS}(\llbracket v_3 \rrbracket, \llbracket p_3 \rrbracket, \llbracket z \rrbracket, \ell + 1, g)$;
- 5 **return** $(\llbracket v \rrbracket, \llbracket p \rrbracket, \llbracket s \rrbracket, \llbracket z \rrbracket)$;

P 5: NormFLS($(\llbracket v \rrbracket, \llbracket p \rrbracket, \llbracket z \rrbracket, \ell, g)$).

- 1 $\llbracket b \rrbracket \leftarrow \text{LTZ}(\llbracket v \rrbracket - 2^\ell, \ell + 1)$;
- 2 $\llbracket v'' \rrbracket \leftarrow \text{Div2}(\llbracket v \rrbracket, \ell + 1)$;
- 3 $\llbracket v' \rrbracket \leftarrow \llbracket b \rrbracket \llbracket v \rrbracket + (1 - \llbracket b \rrbracket) \llbracket v'' \rrbracket$;
- 4 $\llbracket p' \rrbracket \leftarrow (\llbracket p \rrbracket + 1 - \llbracket b \rrbracket)(1 - \llbracket z \rrbracket) - \llbracket z \rrbracket 2^{g-1}$;
- 5 **return** $(\llbracket v' \rrbracket, \llbracket p' \rrbracket)$;

Floating-point Division. Protocol 6, DivFL, computes $\hat{a} \approx \hat{a}_1 / \hat{a}_2$, for secret $\hat{a}_1, \hat{a}_2, \hat{a} \in \mathbb{Q}_{(\ell, g)}^{FL}$, $\hat{a}_1 = (1 - 2s_1)\bar{v}_1 2^{\bar{p}_1}$, $\hat{a}_2 = (1 - 2s_2)\bar{v}_2 2^{\bar{p}_2}$, $\hat{a} = (1 - 2s)\bar{v} 2^{\bar{p}}$. DivFL divides the significands using secure fixed-point arithmetic and normalizes the result. Let $\bar{v}_1 = \bar{v}_1 2^{-\ell}$, $\bar{v}_2 = \bar{v}_2 2^{-\ell}$ and $\bar{v}_3 = \bar{v}_1 / \bar{v}_2$. Observe that $\bar{v}_1, \bar{v}_2 \in [0.5, 1) \cup \{0\}$, $\bar{v}_2 \neq 0$ and $\bar{v}_3 \in (0.5, 2) \cup \{0\}$. Step 1 computes \bar{v}_3 using Protocol 7, DivGS, and steps 2-3 compute $\bar{p}_3 = \bar{p}_1 - \bar{p}_2 - \ell$ and $s = s_1 \oplus s_2$ ($z = z_1$). DivGS returns $\bar{v}_3 \in [2^{\ell-1}, 2^{\ell+1} - 1] \cup \{0\}$ and $\bar{v}_3 = \bar{v}_3 2^\ell$, so we can use NormFLS for normalization.

The protocol DivGS is based on a variant of Goldschmidt's division algorithm (Markstein, 2004). Let $a, b \in \mathbb{R}$, $b \neq 0$. The algorithm starts with an initial approximation $w_0 \approx 1/b$, with relative error $\epsilon_0 < 1$, and computes a/b iteratively, as follows: $c_0 = aw_0$, $d_0 = \epsilon_0 = 1 - bw_0$; for $i > 0$ do $c_i = c_{i-1}(1 + d_{i-1})$, $d_i = d_{i-1}^2$. After i iterations it obtains $c_i \approx a/b$ with relative error ϵ_0^i . If $b \in [0.5, 1)$, we can start with $w_0 = 2.9142 - 2b$, a linear approximation of $1/b$ with relative error $\epsilon_0 < 0.08578$ (Ercegovic and Lang, 2003). It provides about 3.5 exact bits, so for ℓ -bit inputs the algorithm needs $\theta = \lceil \log_{3.5} \frac{\ell}{3.5} \rceil$ iterations.

DivGS uses this algorithm to compute $\bar{v}_3 \approx \bar{v}_1 / \bar{v}_2$ with absolute error $\delta < 2^{-\ell}$, for $\bar{v}_1, \bar{v}_2 \in [0.5, 1) \cup \{0\}$, $\bar{v}_2 \neq 0$, and $\bar{v}_3 \in [0.5, 2) \cup \{0\}$. The inputs and the output are fixed-point numbers with resolution $2^{-\ell}$, encoded as $\bar{v}_1, \bar{v}_2 \in [2^{\ell-1}, 2^\ell - 1] \cup \{0\}$, $\bar{v}_2 \neq 0$, and $\bar{v}_3 \in [2^{\ell-1}, 2^{\ell+1} - 1] \cup \{0\}$. Fixed-point multiplication with resolution $2^{-\ell}$ is computed as double-precision integer multiplication followed by truncation that cuts

off the least significant ℓ bits. The rounding error due to truncation is $\delta_r < 2^{-\ell}$.

P 6: $\text{DivFL}(\{\llbracket v_i \rrbracket, \llbracket p_i \rrbracket, \llbracket s_i \rrbracket, \llbracket z_i \rrbracket\}_{i=1}^2)$.

```

1  $\llbracket v_3 \rrbracket \leftarrow \text{DivGS}(\llbracket v_1 \rrbracket, \llbracket v_2 \rrbracket, \ell)$ ;
2  $\llbracket s \rrbracket \leftarrow \llbracket s_1 \rrbracket \oplus \llbracket s_2 \rrbracket$ ;
3  $\llbracket p_3 \rrbracket \leftarrow \llbracket p_1 \rrbracket - \llbracket p_2 \rrbracket - \ell$ ;
4  $(\llbracket v \rrbracket, \llbracket p \rrbracket) \leftarrow \text{NormFLS}(\llbracket v_3 \rrbracket, \llbracket p_3 \rrbracket, \llbracket z_1 \rrbracket, \ell, g)$ ;
5 return  $(\llbracket v \rrbracket, \llbracket p \rrbracket, \llbracket s \rrbracket, \llbracket z_1 \rrbracket)$ ;

```

P 7: $\text{DivGS}(\llbracket v_1 \rrbracket, \llbracket v_2 \rrbracket, \ell)$.

```

1  $\theta \leftarrow \lceil \log_{\frac{\ell}{3.5}} \rceil$ ;  $m = 4$ ;  $k \leftarrow \ell + m$ ;
2  $\llbracket v_1 \rrbracket \leftarrow 2^m \llbracket v_1 \rrbracket$ ;  $\llbracket v_2 \rrbracket \leftarrow 2^m \llbracket v_2 \rrbracket$ ;
3  $\llbracket w \rrbracket \leftarrow \text{fld}(\text{int}_k(2.9142)) - 2 \llbracket v_2 \rrbracket$ ;
4  $\llbracket c \rrbracket \leftarrow \text{Div2mPD}(\llbracket v_1 \rrbracket * \llbracket w \rrbracket, 2k + 1, k)$ ;
5  $\llbracket d \rrbracket \leftarrow \text{Div2mPD}(\llbracket v_2 \rrbracket * \llbracket w \rrbracket, 2k + 1, k)$ ;
6  $\llbracket d \rrbracket \leftarrow \text{fld}(\text{int}_k(1.0)) - \llbracket d \rrbracket$ ;
7 foreach  $i \in [1, \theta - 1]$  do
8    $\llbracket c \rrbracket \leftarrow \llbracket c \rrbracket + \text{Div2mPD}(\llbracket c \rrbracket * \llbracket d \rrbracket, 2k + 1, k)$ ;
9    $\llbracket d \rrbracket \leftarrow \text{Div2mPD}(\llbracket d \rrbracket * \llbracket d \rrbracket, 2k + 1, k)$ ;
    $\llbracket d \rrbracket \leftarrow \llbracket d \rrbracket$ ;
10  $\llbracket v_3 \rrbracket \leftarrow \llbracket c \rrbracket + \text{Div2mPD}(\llbracket c \rrbracket * \llbracket d \rrbracket, 2k + 1, k + m)$ ;
11 return  $\llbracket v_3 \rrbracket$ ;

```

We prefer this algorithm to other variants (e.g., Newton-Raphson) because the two multiplications of an iteration can be computed in parallel. On the other hand, its iterations are not self-correcting, so rounding errors accumulate, reducing the accuracy of the result. Moreover, if the error before the last truncation is $|\delta| \geq 2^{-\ell}$, \tilde{v}_3 may be outside the range $[2^{\ell-1}, 2^{\ell+1} - 1] \cup \{0\}$ required by fast normalization with NormFLS. For instance, if $\tilde{v}_1 = 2^{\ell-1}$ and $\tilde{v}_2 = 2^\ell - 1$ the output can be $\tilde{v}_3 < 2^{\ell-1}$ ($\tilde{v}_1 = 0.5$, $\tilde{v}_2 \approx 1$, $\tilde{v}_3 \approx 0.5$); also, if $\tilde{v}_1 = 2^\ell - 1$ and $\tilde{v}_2 = 2^{\ell-1}$, the output can be $\tilde{v}_3 > 2^{\ell+1} - 1$ ($\tilde{v}_1 \approx 1$, $\tilde{v}_2 = 0.5$, $\tilde{v}_3 \approx 2$).

Let Δ be the accumulated error before the last truncation and suppose $\Delta < \gamma \cdot 2^{-\ell}$ for variables with ℓ -bit fractional part. The error can be reduced by terminating the algorithm with a modified Newton-Raphson iteration (Markstein, 2004); this requires additional rounds. DivGS reduces the error to $\Delta < 2^{-\ell}$ by increasing the fractional part to $\ell + m$ bits, with $m = \lceil \log \gamma \rceil$. For our initial approximation, error analysis shows that we need $m = 3$ for $\ell \in [8, 14]$ ($\theta = 2$) and $m = 4$ for $\ell \in [15, 112]$ ($\theta \in [3, 5]$). For simplicity, we set $m = 4$ in the pseudocode².

DivGS computes Goldschmidt's iterations for secret inputs and outputs. Steps 1-3 initialize the algorithm: compute θ , m , and $k = \ell + m$; set $\tilde{v}_1 = \tilde{v}_1 2^m$ and $\tilde{v}_2 = \tilde{v}_2 2^m$ to obtain fixed-point numbers with frac-

²The error bound is computed starting from $c_\theta = c_0(1 + d_0)(1 + d_1) \dots (1 + d_{\theta-1})$ and assuming $\delta_r = 2^{-\ell}$ for every multiplication, including $c_0 = aw_0$ and $d_0 = 1 - bw_0$.

tional part of $\ell + m$ bits; compute $\tilde{w} = \tilde{\beta} - 2\tilde{v}_2$, the initial approximation of $1/\tilde{v}_2$. Steps 4-6 compute in parallel the initial values for the iteration variables: $\tilde{c} = \tilde{v}_1 \tilde{w}/2^k$ and $\tilde{d} = (1 - \tilde{v}_2)\tilde{w}/2^k$. Steps 7-10 are the θ iterations of the algorithm. An iteration computes in parallel $\tilde{c} = \tilde{c} + \lfloor (\tilde{c}\tilde{d})/2^k \rfloor$ and $\tilde{d}' \leftarrow \lfloor \tilde{d}^2/2^k \rfloor$ and then sets $\tilde{d} = \tilde{d}'$. The result is in the interval required for fast normalization regardless of the rounding method of the last truncation, so we can use Div2mP.

The online complexity of DivFL is $5 + \theta$ rounds and $\ell + 2\theta + 7$ interactive operations (e.g., 9 rounds and $\ell + 16$ operations for $\ell \in [29, 56]$). DivFL is more accurate and more efficient than the protocol given in (Aliasgari et al., 2013), which does not address the critical accuracy issues discussed above and needs $2 \log \ell + 7$ rounds and $2(\ell + 2) \log \ell + 3\ell + 8$ operations. The complexity improvement is due to better initial approximation (less iterations) and more efficient secure fixed-point arithmetic.

An alternative approach to floating-point division with secret inputs and output, suggested in related work, is to first compute the reciprocal $\hat{a}'_2 = 1/\hat{a}_2$ and then $\hat{a}_3 = \hat{a}_1 \cdot \hat{a}'_2$. However, DivFL has the same complexity as a protocol that computes $1/\hat{a}_2$ and avoids the additional secure floating-point multiplication. Also, with minor changes, DivFL can compute $\hat{a}_3 = \hat{a}_1/\hat{a}_2$ for public \hat{a}_1 and secret \hat{a}_2 and \hat{a}_3 , with slightly lower complexity. Finally, for public \hat{a}_2 and secret \hat{a}_1 and \hat{a}_3 , division consists of secure multiplication between \hat{a}_1 and public $1/\hat{a}_2$.

Square Root. Protocol 8, SqrtFL, computes $\hat{a} \approx \sqrt{|\hat{a}_1|}$, for secret $\hat{a}_1, \hat{a} \in \mathbb{Q}_{(\ell, g)}^{FL}$, $|\hat{a}_1| = \tilde{v}_1 2^{\tilde{p}_1}$ and $\hat{a} = \tilde{v}_2 2^{\tilde{p}}$. SqrtFL is similar to DivFL and surprisingly efficient. The computation is based on the following remark. Let $\tilde{v}_1 = \tilde{v}_1 2^{-\ell} \in [0.5, 1) \cup \{0\}$, encoded as $\tilde{v}_1 \in \mathbb{Q}_{(\ell, \ell)}^{FX}$. Also, let $\tilde{p}'_1 = \tilde{p}_1 + \ell$, $u = \tilde{p}'_1 \bmod 2$, and $\tilde{p}_2 = \lfloor \tilde{p}'_1/2 \rfloor$. Observe that $\sqrt{|\hat{a}_1|} = \sqrt{\tilde{v}_1 2^{\tilde{p}_1 + \ell}}$, so if $u = 0$ then $\sqrt{|\hat{a}_1|} = \sqrt{\tilde{v}_1} 2^{\tilde{p}_2}$, otherwise $\sqrt{|\hat{a}_1|} = \sqrt{2\tilde{v}_1} 2^{\tilde{p}_2} = \frac{\sqrt{2}}{2} \sqrt{\tilde{v}_1} 2^{\tilde{p}_2 + 1}$.

SqrtFL computes $\sqrt{\tilde{v}_1}$ using Protocol 9, SqrtGS, based on Goldschmidt's square root algorithm (Markstein, 2004). Let $a \in \mathbb{R}$, $a > 0$, and $w_0 \approx \frac{1}{\sqrt{a}}$ such that $aw_0^2 \in [\frac{1}{2}, \frac{3}{2}]$. The algorithm computes both \sqrt{a} and $\frac{1}{2\sqrt{a}}$ iteratively, as follows: $b_0 = aw_0$, $c_0 = w_0/2$; for $i > 0$ do $d_{i-1} = 0.5 - b_{i-1}c_{i-1}$, $b_i = b_{i-1}(1 + d_{i-1})$, $c_i = c_{i-1}(1 + d_{i-1})$. After i iterations it obtains $b_i \approx \sqrt{a}$ and $c_i \approx \frac{1}{2\sqrt{a}}$ with relative error ϵ_0^i . If $a \in [0.5, 1)$, we can take $w_0 = 1.7877 - 0.81a$, a linear approximation of $\frac{1}{\sqrt{a}}$ with relative error $\epsilon_0 < 0.0223$. Since w_0 provides almost 5.5 exact bits, the algorithm needs $\theta = \lceil \log_{\frac{\ell}{5.5}} \rceil$ iterations for an ℓ -bit input.

SqrtGS computes $\tilde{b} \approx \sqrt{\tilde{v}}$ for $\tilde{v} \in [0.5, 1) \cup \{0\}$ and $\tilde{b} \in [\frac{\sqrt{2}}{2}, 1) \cup \{0\}$ using secure fixed-point arithmetic. Rounding errors are handled like in DivGS, by extending the fractional part to $k = \ell + m$ bits. Steps 2-3 compute $\tilde{w} = 1.7877 - 0.81\tilde{v}$, the linear approximation of $\frac{1}{\sqrt{\tilde{v}}}$. Steps 4-5 compute (in parallel) the initial values of the variables, $\tilde{b} = \lfloor \tilde{v}\tilde{w}/2^k \rfloor$ and $\tilde{c} = \lfloor \tilde{v}/2 \rfloor$. The steps 6-11 are the θ iterations of the algorithm. An iteration computes $\tilde{d} = \text{int}_k(0.5) - \lfloor \tilde{b}\tilde{c}/2^k \rfloor$ and then $\tilde{b} = \tilde{b} + \lfloor \tilde{b}\tilde{d}/2^k \rfloor$ and $\tilde{c} = \tilde{c} + \lfloor \tilde{c}\tilde{d}/2^k \rfloor$ (steps 8-9 in parallel). The output preserves the higher precision, so that SqrtFL can accurately compute $\frac{\sqrt{2}}{2}\sqrt{\tilde{v}}$.

P 8: SqrtFL($\llbracket v_1 \rrbracket, \llbracket p_1 \rrbracket, \llbracket z_1 \rrbracket$).

```

1  $m = 4; k \leftarrow \ell + m;$ 
2  $\llbracket v_2 \rrbracket \leftarrow \text{SqrtGS}(2^m \llbracket v_1 \rrbracket, \ell, k);$ 
3  $\llbracket v'_2 \rrbracket \leftarrow \text{Div2mP}(\text{fld}(\text{int}_k(\sqrt{2}/2)) \llbracket v_2 \rrbracket, 2k, k + m);$ 
4  $\llbracket v_2 \rrbracket \leftarrow \text{Div2mP}(\llbracket v_2 \rrbracket, k + m, m);$ 
5  $\llbracket p'_1 \rrbracket \leftarrow \llbracket p_1 \rrbracket + \ell; \llbracket p_2 \rrbracket \leftarrow \text{Div2}(\llbracket p'_1 \rrbracket, g + 1);$ 
6  $\llbracket u \rrbracket \leftarrow \llbracket p'_1 \rrbracket - 2 \llbracket p_2 \rrbracket;$ 
7  $\llbracket v \rrbracket \leftarrow (1 - \llbracket u \rrbracket) \llbracket v_2 \rrbracket + \llbracket u \rrbracket \llbracket v'_2 \rrbracket;$ 
8  $\llbracket p \rrbracket \leftarrow (\llbracket p_2 \rrbracket - \ell + \llbracket u \rrbracket)(1 - \llbracket z_1 \rrbracket) - 2^{g-1} \llbracket z_1 \rrbracket;$ 
9 return ( $\llbracket v \rrbracket, \llbracket p \rrbracket, \llbracket z_1 \rrbracket$ );
```

P 9: SqrtGS($\llbracket v \rrbracket, \ell, k$).

```

1  $\theta \leftarrow \lceil \log_{\frac{\ell}{5.5}} \rceil; \alpha \leftarrow \text{fld}(\text{int}_k(0.5));$ 
2  $\llbracket w \rrbracket \leftarrow \text{Div2mP}(\text{fld}(\text{int}_k(0.81)) \llbracket v \rrbracket, 2k, k);$ 
3  $\llbracket w \rrbracket \leftarrow \text{fld}(\text{int}_k(1.7877)) - \llbracket w \rrbracket;$ 
4  $\llbracket b \rrbracket \leftarrow \text{Div2mPD}(\llbracket v \rrbracket * \llbracket w \rrbracket, 2k + 1, k);$ 
5  $\llbracket c \rrbracket \leftarrow \text{Div2}(\llbracket w \rrbracket, k + 1);$ 
6 foreach  $i \in [1, \theta - 1]$  do
7    $\llbracket d \rrbracket \leftarrow \alpha - \text{Div2mPD}(\llbracket b \rrbracket * \llbracket c \rrbracket, 2k + 1, k);$ 
8    $\llbracket b \rrbracket \leftarrow \llbracket b \rrbracket + \text{Div2mPD}(\llbracket b \rrbracket * \llbracket d \rrbracket, 2k + 1, k);$ 
9    $\llbracket c \rrbracket \leftarrow \llbracket c \rrbracket + \text{Div2mPD}(\llbracket c \rrbracket * \llbracket d \rrbracket, 2k + 1, k);$ 
10  $\llbracket d \rrbracket \leftarrow \alpha - \text{Div2mPD}(\llbracket b \rrbracket * \llbracket c \rrbracket, 2k + 1, k);$ 
11  $\llbracket b \rrbracket \leftarrow \llbracket b \rrbracket + \text{Div2mPD}(\llbracket b \rrbracket * \llbracket d \rrbracket, 2k + 1, k);$ 
12 return ( $\llbracket b \rrbracket$ );
```

SqrtFL computes the square root of $|\hat{a}_1|$ as follows. Let $\tilde{v}_2 \approx \sqrt{\tilde{v}_1} \in [\frac{\sqrt{2}}{2}, 1) \cup \{0\}$ and $\tilde{v}'_2 \approx \frac{\sqrt{2}}{2}\tilde{v}_2 \in [0.5, \frac{\sqrt{2}}{2}) \cup \{0\}$, encoded as $\tilde{v}_2, \tilde{v}'_2 \in \mathbb{Q}_{(k,k)}^{FX}$. Steps 2-4 compute \tilde{v}_2 and \tilde{v}'_2 using SqrtGS, and steps 5-6 compute $\tilde{p}'_1, \tilde{p}_2$, and u . If $u = 0$ then $\sqrt{|\hat{a}_1|} = \tilde{v}_2 2^{\tilde{p}_2}$, so we set $\tilde{v} = \lfloor \tilde{v}_2/2^m \rfloor$ and $\tilde{p} = \tilde{p}_2 - \ell$. Otherwise, $\sqrt{|\hat{a}_1|} = \tilde{v}'_2 2^{\tilde{p}_2+1}$; we compute $\tilde{\mu} = \text{int}_k(\frac{\sqrt{2}}{2})$ and $\tilde{v}'_2 = (\tilde{\mu}\tilde{v}_2)/2^k$ and set $\tilde{v} = \lfloor \tilde{v}'_2/2^m \rfloor$ and $\tilde{p} = \tilde{p}_2 - \ell + 1$. The two cases are obviously computed in steps 7-8: $\tilde{v} = (1 - u)\tilde{v}_2 + u\tilde{v}'_2$ and $\tilde{p} = \tilde{p}_2 - \ell + u$. The result is already normalized.

The online complexity of SqrtFL is $4 + 2\theta$ rounds and, surprisingly, only $3\theta + 7$ interactive operations

(e.g., $\theta = 3$ for $\ell \in [24, 45]$). SqrtFL is much more efficient than the protocol suggested in (Aliasgari et al., 2013), that computes Goldschmidt's iterations using floating-point protocols.

Floating-point comparison. Protocol 10, LTFL, computes $c = (\hat{a}_1 < \hat{a}_2)? 1 : 0$ for $\hat{a}_1, \hat{a}_2 \in \mathbb{Q}_{(\ell,g)}^{FL}$, $\hat{a}_1 = (1 - s_1)\tilde{v}_1 2^{\tilde{p}_1}$ and $\hat{a}_2 = (1 - s_2)\tilde{v}_2 2^{\tilde{p}_2}$, with secret inputs and output. The protocol is based on the following idea. Let $\tilde{v}'_1 = (1 - s_1)\tilde{v}_1, \tilde{v}'_2 = (1 - s_2)\tilde{v}_2$ and $\hat{d} = \hat{a}_1 - \hat{a}_2 = 2^{\tilde{p}_2}(\tilde{v}'_1 2^{\tilde{p}_1 - \tilde{p}_2} - \tilde{v}'_2)$. We want to compute $c = (\hat{d} < 0)? 1 : 0$. Also, let $z_p = (\tilde{p}_1 = \tilde{p}_2)? 1 : 0$, $c_p^- = (\tilde{p}_1 < \tilde{p}_2)? 1 : 0$, $c_p^+ = (\tilde{p}_1 > \tilde{p}_2)? 1 : 0$ and $c_v^- = (\tilde{v}'_1 < \tilde{v}'_2)? 1 : 0$. Observe that $\hat{d} < 0$ if and only if one of the following mutually exclusive conditions holds: $\tilde{p}_1 = \tilde{p}_2$ and $\tilde{v}_1 < \tilde{v}_2$; $\tilde{p}_1 < \tilde{p}_2$ and $s_2 = 0$; $\tilde{p}_1 > \tilde{p}_2$ and $s_1 = 1$. Therefore, the output is $c = z_p c_v^- + c_p^-(1 - s_2) + c_p^+ s_1$ (inner product).

We could compute c_p^- and z_p , using the protocols LTZ and EQZ (Catrina and de Hoogh, 2010a), and then $c_p^+ = (1 - c_p^-)(1 - z_p)$. Instead, we introduce Protocol 11, CmpZ, that computes more efficiently the triple comparison. Thus, we obtain a simpler and more efficient solution for LTFL: steps 1-2 compute c_v^- using LTZD, step 3 computes c_p^-, c_p^+ and z_p using CmpZ, and step 4 computes the output.

P 10: LTFL($\{\llbracket v_i \rrbracket, \llbracket p_i \rrbracket, \llbracket s_i \rrbracket\}_{i=1}^2$).

```

1  $\llbracket d \rrbracket_{2t} \leftarrow (1 - 2 \llbracket s_1 \rrbracket) * \llbracket v_1 \rrbracket - (1 - 2 \llbracket s_2 \rrbracket) * \llbracket v_2 \rrbracket;$ 
2  $\llbracket c_v^- \rrbracket \leftarrow \text{LTZD}(\llbracket d \rrbracket_{2t}, \ell + 1);$ 
3  $\llbracket c_p^- \rrbracket, \llbracket c_p^+ \rrbracket, \llbracket z_p \rrbracket \leftarrow \text{CmpZ}(\llbracket p_1 \rrbracket - \llbracket p_2 \rrbracket, g + 1);$ 
4  $\llbracket c \rrbracket \leftarrow \llbracket z_p \rrbracket \llbracket c_v^- \rrbracket + \llbracket c_p^- \rrbracket (1 - \llbracket s_2 \rrbracket) + \llbracket c_p^+ \rrbracket \llbracket s_1 \rrbracket;$ 
5 return ( $\llbracket c \rrbracket$ );
```

P 11: CmpZ($\llbracket a \rrbracket, k$).

```

1 ( $\llbracket r'' \rrbracket, \llbracket r' \rrbracket, \{\llbracket r'_i \rrbracket\}_{i=1}^{k-1}$ )  $\leftarrow \text{PRandM}(k, k - 1);$ 
2  $b \leftarrow \text{Reveal}(2^{k-1} + \llbracket a \rrbracket + 2^{k-1} \llbracket r'' \rrbracket + \llbracket r' \rrbracket);$ 
3  $b' \leftarrow b \bmod 2^{k-1};$ 
4 ( $\llbracket u_1 \rrbracket, \llbracket u_2 \rrbracket$ )  $\leftarrow \text{BitCmp}(b', \{\llbracket r'_i \rrbracket\}_{i=1}^{k-1});$ 
5  $\llbracket c_1 \rrbracket \leftarrow -((\llbracket a \rrbracket - (b' - \llbracket r' \rrbracket))2^{-(k-1)} - \llbracket u_1 \rrbracket);$ 
6  $\llbracket c_2 \rrbracket \leftarrow (1 - \llbracket c_1 \rrbracket)(1 - \llbracket u_2 \rrbracket);$ 
7  $\llbracket c_3 \rrbracket \leftarrow (1 - \llbracket c_1 \rrbracket) \llbracket u_2 \rrbracket;$ 
8 return ( $\llbracket c_1 \rrbracket, \llbracket c_2 \rrbracket, \llbracket c_3 \rrbracket$ );
```

Given a secret integer $\bar{a} \in \mathbb{Z}_{(k)}$, CmpZ returns the secret bits $c_1 = (\bar{a} < 0)? 1 : 0$, $c_2 = (\bar{a} > 0)? 1 : 0$, and $c_3 = (\bar{a} = 0)? 1 : 0$. CmpZ uses Protocol 12, BitCmp, with input a public integer $\bar{a} = \sum_{i=1}^k 2^{i-1} a_i$ and a bitwise-shared integer $\bar{b} = \sum_{i=1}^k 2^{i-1} b_i$, and output the secret bits $u_1 = (\bar{a} < \bar{b})? 1 : 0$ and $u_2 = (\bar{a} = \bar{b})? 1 : 0$.

CmpZ extends the protocol LTZ to compute the bits c_2 and c_3 , besides c_1 . Steps 1-5 compute $c_1 =$

$-\lfloor \bar{a}/2^{k-1} \rfloor$ exactly like LTZ, except that BitLT is replaced by BitCmp in step 4. Steps 1-3 compute and reveal $b = 2^{k-1} + \bar{a} + r$, where $r = 2^{k-1}r'' + r'$ is a random secret integer that hides \bar{a} with statistical secrecy and $r' = \sum_{i=1}^{k-1} 2^{i-1}r_i$, with $\{r'_i\}_{i=1}^{k-1}$ uniformly random secret bits. Let $b' = b \bmod 2^{k-1}$ and $a' = \bar{a} \bmod 2^{k-1}$. Step 4 computes $u_1 = (b' < r')? 1 : 0$ and $u_2 = (b' = r')? 1 : 0$. Observe that $b' = a' + r' - 2^{k-1}u_1$, so $\lfloor \bar{a}/2^{k-1} \rfloor = (\bar{a} - (b' - r'))2^{-(k-1)} - u_1$. Also, $u_2 = 1$ if $\bar{a} = 0$ or $\bar{a} = -2^{k-1}$, so $c_2 = (1 - c_1)(1 - u_2)$ and $c_3 = (1 - c_1)u_2$ (steps 6-7, in parallel).

P 12: BitCmp($a, \{\llbracket b_i \rrbracket\}_{i=1}^k$).

```

1 foreach  $i \in [1, k]$  do  $\llbracket d_i \rrbracket \leftarrow a_i \oplus \llbracket b_i \rrbracket$ ;
2 foreach  $i \in [1, k]$  do  $c_i \leftarrow 1 - a_i$ ;
3  $\{\llbracket p_i \rrbracket\}_{i=1}^k \leftarrow \text{SufMul}(\{\llbracket d_i + 1 \rrbracket\}_{i=1}^k)$ ;
4  $\llbracket s_1 \rrbracket \leftarrow c_k \llbracket d_k \rrbracket + \sum_{i=1}^{k-1} c_i (\llbracket p_i \rrbracket - \llbracket p_{i+1} \rrbracket)$ ;
5  $\llbracket u_1 \rrbracket \leftarrow \text{Mod2}(\llbracket s_1 \rrbracket, k)$ ;
6  $\llbracket u_2 \rrbracket \leftarrow \text{Mod2}(\llbracket p_1 \rrbracket, k)$ ;
7 return  $(\llbracket u_1 \rrbracket, \llbracket u_2 \rrbracket)$ ;
```

BitCmp is similar to the protocol BitLT given in (Catrina and de Hoogh, 2010a). Steps 1-5 compute u_1 exactly like BitLT, so we explain only the computation of u_2 . Step 3 computes $p_i = \prod_{j=i}^k (d_j + 1)$, for $i \in [1, k]$, where $d_j = a_j \oplus b_j$. If $a = b$ then $p_1 = 1$, else p_1 is a power of 2, so $u_2 = p_1 \bmod 2$. Steps 5-6 run in parallel, so we obtain u_2 almost for free.

LTFL can also be used to compute the other comparison operators, by observing that $c = (\hat{a}_1 < \hat{a}_2)? 1 : 0 = (\hat{a}_2 > \hat{a}_1)? 1 : 0$ and $1 - c = (\hat{a}_1 \geq \hat{a}_2)? 1 : 0 = (\hat{a}_2 \leq \hat{a}_1)? 1 : 0$. Moreover, it also works when an operand is public, with the same complexity.

The online complexity of BitCmp is 2 rounds and $k + 2$ interactive operations (1 operation more than BitLT) and CmpZ needs 4 rounds and $k + 5$ interactive operations. Therefore, the online complexity of LTFL is 5 rounds and $\ell + g + 7$ interactive operations (steps 2-3 in parallel). This is similar to comparison in $\mathbb{Q}_{(k,f)}^{FX}$ using LTZ (LTFL adds 2 rounds, but usually $\ell + g < k$, so the communication complexity is lower).

Equality of secret $\hat{a}_1, \hat{a}_2 \in \mathbb{Q}_{(\ell,g)}^{FL}$ with secret output can be tested as efficiently as for fixed-point numbers, based on the following remark. Let $\hat{a}_1 = (1 - s_1)\bar{v}_1 2^{\bar{p}_1}$, $\hat{a}_2 = (1 - s_2)\bar{v}_2 2^{\bar{p}_2}$, and $c = (\hat{a}_1 = \hat{a}_2)? 1 : 0$. Also, let $\Delta = 2^{\ell+1}(\bar{p}_1 - \bar{p}_2) + 2^\ell(\bar{s}_1 - \bar{s}_2) + (\bar{v}_1 - \bar{v}_2) = 2^{\ell+1}\bar{d}_p + 2^\ell\bar{d}_s + \bar{d}_v$. Observe that $\bar{d}_s \in \{-1, 0, 1\}$ and $|\bar{d}_v| < 2^\ell$. If $\bar{d}_s \neq 0$ and $\bar{d}_p \neq 0$ then $0 < |2^\ell\bar{d}_s + \bar{d}_v| < |2^{\ell+1}\bar{d}_p|$, hence $\Delta \neq 0$. Thus, $\Delta = 0$ if and only if $\bar{d}_p = 0$, $\bar{d}_s = 0$, and $\bar{d}_v = 0$, hence $c = (\Delta = 0)? 1 : 0$.

Protocol 13, EQFL, computes $c = (\hat{a}_1 = \hat{a}_2)? 1 : 0$ as described above. Its online complexity is 3 rounds and $\log(\ell + g + 2) + 2$ interactive operations, the same

as for inputs in $\mathbb{Q}_{(k,f)}^{FX}$ with $k > \ell + g$.

P 13: EQFL($\{\llbracket v_i \rrbracket, \llbracket p_i \rrbracket, \llbracket s_i \rrbracket\}_{i=1}^2$).

```

1  $\llbracket b_1 \rrbracket \leftarrow 2^{\ell+1} \llbracket p_1 \rrbracket + 2^\ell \llbracket s_1 \rrbracket + \llbracket v_1 \rrbracket$ ;
2  $\llbracket b_2 \rrbracket \leftarrow 2^{\ell+1} \llbracket p_2 \rrbracket + 2^\ell \llbracket s_2 \rrbracket + \llbracket v_2 \rrbracket$ ;
3  $\llbracket c \rrbracket \leftarrow \text{EQZ}(\llbracket b_1 \rrbracket - \llbracket b_2 \rrbracket, \ell + g + 2)$ ;
4 return  $\llbracket c \rrbracket$ ;
```

4 CONCLUSIONS

A broad range of privacy preserving collaborative applications require efficient secure computation with real numbers (statistical analysis, benchmarking, data mining, and optimizations). Starting from the framework introduced in (Catrina and de Hoogh, 2010a; Catrina and Saxena, 2010), we add building blocks and optimizations that alleviate the performance bottlenecks of the previous protocols. We show that secure floating-point arithmetic is substantially improved using a small set of powerful and efficient building blocks (Table 2) and protocol constructions.

The online and precomputation complexity of the floating-point protocols is summarized in Table 3 (with $\theta_1, \theta_2 \in [3, 4]$ for $\ell \in [24, 56]$). All protocols are specified for secret operands and secret result, in the same security model. However, they can be adapted to also work when one of the operands is public. In some cases, the complexity is significantly lower when part of the input information is not secret, e.g., FX2FL for input with known sign, AddFL for operands with the same sign, DivFL with public divisor.

A challenge for secure arithmetic is to find the best complexity trade-offs, taking into account a complete protocol family and typical applications³. We analyzed several variants of floating-point encoding and building blocks. The selection presented in the paper offers better tradeoffs for the round and communication complexity of the entire protocol family. The additional building blocks (PreDiv2m, PreDiv2mP, Int2MaskG, and non-interactive multiplication) improve the performance of the underlying integer and fixed-point arithmetic protocols and allow us to use simpler algorithms in the floating-point protocols.

We focused on performance, accuracy and flexibility, rather than trying to replicate the format and features specified in the IEEE Standard for Floating-Point Arithmetic (IEEE 754). The parameters ℓ and

³Sign and magnitude encoding of the significand offers better tradeoffs. A compact encoding $\langle \bar{v}, \bar{p} \rangle$, $\hat{a} = \bar{v} 2^{\bar{p}}$, $\bar{v} \in [2^{\ell-2}, 2^{\ell-1} - 1] \cup [-2^{\ell-1}, -2^{\ell-2} - 1] \cup \{0\}$ simplifies FX2FL and AddFL, but complicates the other operations. Also, if we remove z and ignore \bar{p} when $\bar{v} = 0$, we have to compute z and $\bar{p} = (1 - z)\bar{p} - z2^{\bar{s}-1}$ in AddFL (4 rounds).

Table 3: Complexity of floating-point protocols for $\hat{a}, \hat{a}_i \in \mathbb{Q}_{(\ell, g)}^{FL}$, $\tilde{a} \in \mathbb{Q}_{(k, f)}^{FX}$.

Protocol	Task	Rounds	Inter. op.	Prec.
LTFL	$(\hat{a}_1 < \hat{a}_2)? 1 : 0$	5	$\ell + g + 7$	$3(\ell + g)$
EQFL	$(\hat{a}_1 = \hat{a}_2)? 1 : 0$	3	$\log(\ell + g) + 2$	$\ell + g + 3 \log(\ell + g)$
FX2FL	$\hat{a} \leftarrow \tilde{a}$	9	$5k + 3$	$10k - 11$
AddFL	$\hat{a} \leftarrow \hat{a}_1 + \hat{a}_2$	19	$6\ell + 2g + 26$	$\approx 13\ell + 9g$
MulFL	$\hat{a} \leftarrow \hat{a}_1 \hat{a}_2$	5	$\ell + 9$	$4\ell + 6$
DivFL	$\hat{a} \leftarrow \hat{a}_1 / \hat{a}_2$	$5 + \theta_1$	$\ell + 2\theta_1 + 6$	$\approx (2\theta_1 + 4)\ell$
SqrtFL	$\hat{a} \leftarrow \sqrt{\hat{a}_1}$	$4 + 2\theta_2$	$3\theta_2 + 7$	$\approx (3\theta_2 + 3)\ell$

Table 4: Running time of floating-point protocols (milliseconds/operation).

Batch size	1	Prec.	10	Prec.	20	Prec.	50	Prec.	100	Prec.
LTFL	1.88	1.83	0.46	1.08	0.39	1.10	0.30	1.04	0.27	1.04
EQFL	0.94	1.35	0.17	0.71	0.13	0.70	0.09	0.64	0.08	0.63
MulFL	1.89	1.20	0.42	1.19	0.32	1.18	0.24	1.12	0.22	1.11
DivFL	3.09	4.47	0.53	3.56	0.42	3.63	0.32	3.57	0.32	3.59
SqrtFL	3.28	4.28	0.45	3.64	0.32	3.72	0.24	3.64	0.22	3.67
AddFL	7.98	5.60	2.66	4.39	2.37	4.38	2.14	4.23	1.96	4.18

g determine the range and precision of the floating numbers, as well as the protocols' communication and computation complexity (the size of the field and the number of interactive operations). All protocols take ℓ and g as (implicit) parameters and work accurately, with relative error $2^{-(\ell-1)}$, for the entire range of practically relevant values (including standard simple and double precision). Thus, they can offer the best tradeoff between accuracy and performance, according to application requirements.

The protocols were tested using our Java implementation of the secure computation framework discussed in Section 2. Table 4 shows preliminary performance measurements for 3 parties, $\ell = 32$, $g = 10$, and $\lceil \log q \rceil = 128$. The protocols ran on computers with 3.6 GHz CPU, connected to a 1 Gbps LAN. The results show the baseline performance for low-latency and high-bandwidth networks and single-threaded code. Large batches of primitives can be processed faster by splitting the load among CPU cores. The single-thread code used only a small fraction of the bandwidth⁴. On the other hand, longer network latency means longer interaction rounds. A more comprehensive performance assessment, with broader scope, will be included in future work.

The tests ran the protocols for up to 100 parallel operations. The table lists online and precomputation time per operation. The results of the measurements are well correlated with the complexity, but we expect heavier performance penalty for protocols with larger round complexity in networks with longer transfer de-

⁴The load of the quad-core CPU (Intel i7-7700) was 20% and the data rate was 35-50 Mbps. Tests in a 100 Mbps LAN showed modest performance degradation.

lays. The online time is clearly much shorter when operations are part of larger batches, so the applications that use algorithms with high parallelism will see important performance improvements.

Floating-point arithmetic protocols are inherently more complex than fixed-point arithmetic protocols. This complexity is partially compensated by more compact data encoding: the protocols run more complex algorithms with smaller integers encoded in smaller fields. Multiplication and comparison protocols have similar performance for floating-point and fixed-point numbers, while floating-point division is faster. On the other hand, secure floating-point addition remains complex and relatively slow.

On-going work, being finalized, shows important performance gains for more complex tasks, like evaluating sums and polynomials, by using dedicated protocols, instead of generic constructions. However, these optimized protocols are slower than the fixed-point versions, since adding secret-shared fixed-point numbers is just a local addition of field elements.

This suggests combining secure fixed-point and floating-point arithmetic according to application requirements, an approach we are currently studying.

REFERENCES

- Aliasgari, M., Blanton, M., and Bayatbolghani, F. (2017). Secure Computation of Hidden Markov Models and Secure Floating-point Arithmetic in the Malicious Model. *International Journal of Information Security*, 16(6):577–601.
- Aliasgari, M., Blanton, M., Zhang, Y., and Steele, A. (2013). Secure Computation on Floating Point Num-

- bers. In *20th Annual Network and Distributed System Security Symposium (NDSS'13)*.
- Bogdanov, D., Kamm, L., Laur, S., and Sokk, V. (2018). Rmind: A Tool for Cryptographically Secure Statistical Analysis. *IEEE Transactions On Dependable And Secure Computing*, 15(3):481–495.
- Catrina, O. (2018). Round-Efficient Protocols for Secure Multiparty Fixed-Point Arithmetic. In *12th International Conference on Communications (COMM 2018)*, pages 431–436. IEEE.
- Catrina, O. and de Hoogh, S. (2010a). Improved Primitives for Secure Multiparty Integer Computation. In *Security and Cryptography for Networks*, volume 6280 of *LNCS*, pages 182–199. Springer.
- Catrina, O. and de Hoogh, S. (2010b). Secure Multiparty Linear Programming Using Fixed-Point Arithmetic. In *Computer Security - ESORICS 2010*, volume 6345 of *LNCS*, pages 134–150. Springer.
- Catrina, O. and Saxena, A. (2010). Secure Computation With Fixed-Point Numbers. In *Financial Cryptography and Data Security*, volume 6052 of *LNCS*, pages 35–50. Springer.
- Cramer, R., Damgård, I., and Ishai, Y. (2005). Share Conversion, Pseudorandom Secret-sharing and Applications to Secure Computation. In *Theory of Cryptography (TCC'05)*, volume 3378 of *LNCS*, pages 342–362. Berlin, Heidelberg. Springer.
- Cramer, R., Damgård, I., and Nielsen, J. B. (2015). *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, UK.
- Damgård, I., Fitz, M., Kiltz, E., Nielsen, J. B., and Toft, T. (2006). Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography (TCC 2006)*, volume 3876 of *LNCS*, pages 285–304. Springer.
- Damgård, I. and Thorbek, R. (2007). Non-interactive Proofs for Integer Multiplication. In *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 412–429. Springer.
- Dimitrov, V., Kerik, L., Krips, T., Randmets, J., and Willemson, J. (2016). Alternative Implementations of Secure Real Numbers. In *23rd ACM Conference on Computer and Communications Security (CCS'16)*, pages 553–564. ACM.
- Ercegovic, M. D. and Lang, T. (2003). *Digital Arithmetic*. Morgan Kaufmann.
- Kamm, L. and Willemson, J. (2015). Secure Floating Point Arithmetic and Private Satellite Collision Analysis. *International Journal of Information Security*, 14(6):531–548.
- Krips, T. and Willemson, J. (2014). Hybrid Model of Fixed and Floating Point Numbers in Secure Multiparty Computations. In *Information Security (ISC 2014)*, volume 8783 of *LNCS*, pages 179–197. Springer.
- Markstein, P. (2004). Software Division and Square Root Using Goldschmidt's Algorithms. In *6th Conference on Real Numbers and Computers*, pages 146–157.
- Reistad, T. I. and Toft, T. (2009). Linear, Constant-Rounds Bit-Decomposition. In *International Conference on Information Security and Cryptology*, volume 3329 of *LNCS*, pages 245–257. Springer.

APPENDIX

This appendix provides (for convenience) pseudocode and further details for building blocks presented in previous work and used in this paper.

Given a secret signed integer $\bar{a} \in \mathbb{Z}_{(k)}$ and a public integer $m \in [1, k-1]$, Protocol 14, Div2mP, returns secret $\bar{a}/2^m$ with probabilistic rounding to nearest (Catrina and de Hoogh, 2010a).

Div2mP computes $\bar{c} = \lfloor \bar{a}/2^m \rfloor + u$, for $u \in \{0, 1\}$. Let $\bar{d} = 2^{k-1} + \bar{a}$ and $\bar{d}' = \bar{a} \bmod 2^m$. Observe that $\bar{d} \geq 0$ and $\bar{d} \bmod 2^m = \bar{d}'$ 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, $\bar{c} = (\bar{a} - \bar{d}' + 2^m u) 2^{-m} = \lfloor \bar{a}/2^m \rfloor + u$.

P 14: Div2mP($\llbracket a \rrbracket, k, m$).

```

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

```

P 15: Div2m($\llbracket a \rrbracket, k, m$).

```

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

```

P 16: Div2($\llbracket a \rrbracket, k$).

```

1 ( $\llbracket r'' \rrbracket, \llbracket r' \rrbracket, \llbracket r'_1 \rrbracket$ )  $\leftarrow$  PRandM( $k, 1$ );
2  $b \leftarrow$  Reveal( $2^{k-1} + \llbracket a \rrbracket + 2 \llbracket r'' \rrbracket + \llbracket r'_1 \rrbracket$ );
3  $\llbracket a_1 \rrbracket \leftarrow b_1 + \llbracket r'_1 \rrbracket - 2b_1 \llbracket r'_1 \rrbracket$ ;
4  $\llbracket c \rrbracket \leftarrow (\llbracket a \rrbracket - \llbracket a_1 \rrbracket) 2^{-1}$ ;
5 return  $\llbracket c \rrbracket$ ;

```

Protocol 15, Div2m, is a variant that computes $\bar{a}/2^m$ with deterministic rounding to $-\infty$ (Catrina and de Hoogh, 2010a). This is achieved by computing the bit u using the protocol BitLT. The online complexity of Div2m is 3 rounds and $m+2$ interactive operations. Div2mP needs a single online interaction, so it is much more efficient. However, certain applications require deterministic rounding. Protocol 16, Div2, is a variant of Div2m optimized for $m=1$, that needs a single interactive operation.

Protocol 17, PreDiv2mP, is a generalization of Div2mP that computes $\{\bar{a}'_i\}_{i=1}^m = \{\lfloor \bar{a}/2^i \rfloor\}_{i=1}^m$ with

probabilistic rounding to nearest. Similarly, Protocol 18, PreDiv2m, computes $\{a'_i\}_{i=1}^m = \{\lfloor \bar{a}/2^i \rfloor\}_{i=1}^m$ with deterministic rounding to $-\infty$. PreDiv2m uses the protocol PreBitLT, a generalization of BitLT, to efficiently compute $\{u_i\}_{i=1}^m = \{(b_i < r'_i)? 1 : 0\}_{i=1}^m$.

P 17: PreDiv2mP($\llbracket a \rrbracket, k, m$).

```

1 ( $\llbracket r'' \rrbracket, \llbracket r' \rrbracket, \{\llbracket r'_i \rrbracket\}_{i=1}^m$ )  $\leftarrow$  PRandM( $k, m$ );
2  $b \leftarrow$  Reveal( $2^{k-1} + \llbracket a \rrbracket + 2^m \llbracket r'' \rrbracket + \llbracket r' \rrbracket$ );
3 foreach  $i \in [1, m]$  do
4    $b_i \leftarrow b \bmod 2^i$ ;  $\llbracket s_i \rrbracket \leftarrow \sum_{j=0}^{i-1} 2^j \llbracket r'_j \rrbracket$ ;
5    $\llbracket a'_i \rrbracket \leftarrow (\llbracket a \rrbracket - (b_i - \llbracket s_i \rrbracket))2^{-i}$ ;
6 return  $\{\llbracket a'_i \rrbracket\}_{i=1}^m$ ;
```

P 18: PreDiv2m($\llbracket a \rrbracket, k, m$).

```

1 ( $\llbracket r'' \rrbracket, \llbracket r' \rrbracket, \{\llbracket r'_i \rrbracket\}_{i=1}^m$ )  $\leftarrow$  PRandM( $k, m$ );
2  $b \leftarrow$  Reveal( $2^{k-1} + \llbracket a \rrbracket + 2^m \llbracket r'' \rrbracket + \llbracket r' \rrbracket$ );
3 foreach  $i \in [1, m]$  do
4    $b_i \leftarrow b \bmod 2^i$ ;  $\llbracket s_i \rrbracket \leftarrow \sum_{j=0}^{i-1} 2^j \llbracket r'_j \rrbracket$ ;
5    $\{\llbracket u_i \rrbracket\}_{i=1}^m \leftarrow$  PreBitLT( $b, \{\llbracket r'_i \rrbracket\}_{i=1}^m$ );
6 foreach  $i \in [1, m]$  do
7    $\llbracket a'_i \rrbracket \leftarrow (\llbracket a \rrbracket - (b_i - \llbracket s_i \rrbracket))2^{-i} - \llbracket u_i \rrbracket$ ;
8 return  $\{\llbracket a'_i \rrbracket\}_{i=1}^m$ ;
```

The complexity of PreDiv2m is 3 rounds and $2m + 1$ interactive operations, so it performs a much more complex task than Div2m with the same round complexity and slightly higher communication complexity. PreDiv2mP needs a single online interaction, so we use it instead of PreDiv2m whenever possible.

The efficient solutions described above are based on Protocol 19, BitLT (“bitwise less than”) (Catrina and de Hoogh, 2010a), and its generalization PreBitLT (Catrina, 2018).

BitLT computes the secret bit $u = (a < b)? 1 : 0$ for a non-secret integer $a = \sum_{i=1}^k 2^{i-1} a_i$ and a bitwise-shared integer $b = \sum_{i=1}^k 2^{i-1} b_i$. Steps 1-2 compute $d_i = a_i \oplus b_i$ and the products $p_i = \prod_{j=i}^k (d_j + 1)$, for $i \in [1, k]$; SufMul is the protocol PreMul in (Catrina and de Hoogh, 2010a), with inputs and outputs in inverse order. Observe that $d_j + 1 = 2^{d_j}$, so $p_i = \prod_{j=i}^k 2^{d_j} = 2^{\sum_{j=i}^k d_j}$. Step 3 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 $i \in [m + 1, k]$, so $s = 1 - a_m + 2(1 - a_{m-1})d_{m-1} + 2\sum_{i=1}^{m-2} (1 - a_i)d_i 2^{\sum_{j=i+1}^{m-1} d_j}$. The output is $u = s \bmod 2 = 1 - a_m$. If $m = k$, then $d_k = 1$ 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 output 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 $i \in [1, k]$, so $s = 0$ and the output is $u = 0$.

P 19: BitLT($a, \{\llbracket b_i \rrbracket\}_{i=1}^k$).

```

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

Protocol 20, PreBitLT, is an efficient generalization of BitLT: given a non-secret integer $a = \sum_{i=1}^k 2^{i-1} a_i$ and a bitwise shared integer $b = \sum_{i=1}^k 2^{i-1} b_i$, it computes the secret bits $\{u_i\}_{i=1}^k = \{(a'_i < b'_i)? 1 : 0\}_{i=1}^k$, where $a'_i = \sum_{j=1}^i 2^{j-1} a_j$ and $b'_i = \sum_{j=1}^i 2^{j-1} b_j$ (Catrina, 2018).

P 20: PreBitLT($a, \{\llbracket b_i \rrbracket\}_{i=1}^k$).

```

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

For $u_k = (a'_k < b'_k)? 1 : 0$, 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 $u_\ell = (a'_\ell < b'_\ell)? 1 : 0$, with $\ell \in [1, k - 1]$, PreBitLT computes $s_\ell = \sum_{i=1}^\ell (1 - a_i)(p_i - p_{i+1})$. Since $s_\ell = \sum_{i=1}^\ell (1 - a_i)d_i 2^{\sum_{j=i+1}^\ell d_j}$ 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 computed in \mathbb{Z}_q , as $s_\ell p_{\ell+1}^{-1}$.

SufMullnv computes $\{p_i\}_{i=1}^k = \{\prod_{j=i}^k a_j\}_{i=1}^k$ and $\{p_i^{-1}\}_{i=1}^k = \{\prod_{j=i}^k a_j^{-1}\}_{i=1}^k$. It extends the protocol PreMul and has the same complexity.

The online complexity of BitLT is 2 rounds and $k + 1$ interactive operations. PreBitLT needs 2 rounds and $2k$ interactive operations.