Improved Primitives for Secure Multiparty Integer Computation

Conference Paper · September 2010

DOI: 10.1007/978-3-642-15317-4_13 · Source: DBLP

2 authors:

Octavian Catrina
Polytechnic University of Bucharest
35 PUBLICATIONS **398** CITATIONS

SEE PROFILE

Sebastiaan de Hoogh
Eindhoven University of Technology
9 PUBLICATIONS **264** CITATIONS

SEE PROFILE

Improved Primitives for Secure Multiparty Integer Computation

Octavian Catrina[1] and Sebastiaan de Hoogh[2]

[1] Dept. of Computer Science, University of Mannheim, Germany
[2] Dept. of Mathematics and Computer Science, TU Eindhoven, The Netherlands

Abstract. We consider a collection of related multiparty computation protocols that provide core operations for secure integer and fixed-point computation. The higher-level protocols offer integer truncation and comparison, which are typically the main performance bottlenecks in complex applications. We present techniques and building blocks that allow to improve the efficiency of these protocols, in order to meet the performance requirements of a broader range of applications. The protocols can be constructed using different secure computation methods. We focus on solutions for multiparty computation using secret sharing.

Keywords: Secure multiparty computation, secret sharing, secure integer arithmetic, secure comparison

1 Introduction

The aim of secure computation is to enable a group of mutually distrustful parties to run a joint computation with private inputs. This goal is achieved using cryptographic protocols that carry out the computation without revealing the parties' inputs and ensure that the output is correct.

Applications are found in various areas, including e-voting [10], auctions with secret bids [12], benchmarking with confidential performance indicators [18], collaborative linear programming [25] and supply chain planning [3]. However, the overhead of the cryptographic protocols makes secure computation much slower than usual computation with public data. Improved solutions have emerged for many primitives and application-specific tasks, but meeting the functional, security, and performance requirements of the applications is still a challenge.

In this paper, we focus on improving several related integer computation protocols that support and complement the protocols for fixed-point arithmetic introduced in [6]. The higher level protocols provide accurate truncation (core component for fixed-point arithmetic) and comparison (inequality and equality). These operations are the main performance bottlenecks in complex applications (e.g., multiparty linear programming).

The protocols can be instantiated using secret sharing [8] or homomorphic encryption [9]. We focus on solutions based on secret sharing, semi-honest model, and statistical privacy, which are more suitable for complex applications.

Our contributions. We specify a collection of related protocols for several operations that determine the performance of secure integer and fixed-point computation, namely comparison and truncation. The protocols are based on few building blocks and the same security model, and thus simplify the development of applications and the analysis of their complexity and security. The main goal of the paper is to present techniques and building blocks that reduce the complexity of these protocols in order to meet the requirements of a broader range of applications. We combine several approaches, improving the efficiency of data encoding (adapted to data type), core tasks (generation of secret random values, inner product), and the main building blocks (k-ary and prefix operations, bitwise comparison). Moreover, we give building blocks for different trade-offs between communication and round complexity, so that the protocols can be adapted to applications and execution environment (network bandwidth and delay).

Related Work. We use standard techniques for multiparty computation based on secret sharing, similar to [5,8,11,24,19,4]. Also, the protocols used for generating shared random values and for share conversions rely on techniques proposed in [7,13,14] in order to reduce the communication overhead. We take, however, a more pragmatic approach, allowing more flexibility in the design of the protocols and focusing on solutions that are more suitable for practical applications, while [11,24] aim at achieving perfect privacy and constant round complexity. Protocols with statistical privacy are often more efficient, while those with logarithmic round complexity often have lower communication complexity.

There is a vast literature on secure integer arithmetic. Our approach is related to a large pool of common techniques and protocols [24,11,19,20,15]. We develop more efficient protocols for k-ary and prefix boolean functions, bitwise comparison, and other building blocks. Efficient but approximate truncation is given in [2]; our protocols offer accurate truncation, required by fixed-point arithmetic.

2 Core Protocols

The protocols presented in Sections 3 and 4 are constructed using abstract primitives that can be instantiated using secret sharing [8] or homomorphic encryption [9]. Solutions based on secret sharing are more efficient and suitable for our target applications. Multiparty computation using secret sharing proceeds 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, $i \in [1..n]$, 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. The secret sharing scheme allows to compute with shared variables and provides controlled access to secret values. Sub-tasks take on input shared data and return shared data, and thus enable secure composition.

The protocols offer perfect or statistical privacy, meaning that the views of protocol execution (all values learned 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 and $\Delta(X,Y) = \frac{1}{2}\sum_{v \in V \bigcup W} |Pr(X = v) - Pr(Y = v)|$ the statistical distance between them. 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 core of the system consists of protocols for secure arithmetic in a finite field $\mathbb{F}$ using Shamir secret sharing. These protocols provide perfect privacy against a passive threshold adversary that corrupts 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 uniformly random values in $\mathbb{F}$. We assume $|\mathbb{F}| > n$ to enable Shamir sharing and $n > 2t$ for multiplication of shared values. We denote $[x]^{\mathbb{F}}$ a Shamir sharing of x in field $\mathbb{F}$; if not specified, the field is $\mathbb{Z}_q$.

Complexity metrics. We use two 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 a primitive during which every party sends a share (field element) to the others (Table 1). Round complexity is the number of sequential invocations and is relevant for the network delay, independent of the amount of data. Invocations that can be executed in parallel count as a single round.

Data representation. We consider secure computation with the following data types: binary values, signed integers $\mathbb{Z}_{\langle k \rangle} = \{\bar{x} \in \mathbb{Z} \mid -2^{k-1} \leq \bar{x} \leq 2^{k-1} - 1\}$, and fixed-point rational numbers $\mathbb{Q}_{\langle k,f \rangle} = \{\tilde{x} \in \mathbb{Q} \mid \tilde{x} = \bar{x} \cdot 2^{-f}, \bar{x} \in \mathbb{Z}_{\langle k \rangle}\}$. These data types are encoded in a field $\mathbb{F}$ as follows.

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

Signed integers are encoded in $\mathbb{Z}_q$ using the mapping $\mathsf{fld} : \mathbb{Z}_{\langle k \rangle} \mapsto \mathbb{Z}_q$, $\mathsf{fld}(\bar{x}) = \bar{x} \bmod q$, $q > 2^k$. Secure arithmetic with signed integers is computed using secure arithmetic in $\mathbb{Z}_q$: for any $\bar{a}, \bar{b} \in \mathbb{Z}_{\langle k \rangle}$ and $\odot \in \{+, -, \cdot\}$ we have $\bar{a} \odot \bar{b} = \mathsf{fld}^{-1}(\mathsf{fld}(\bar{a}) \odot \mathsf{fld}(\bar{b}))$; moreover, if $\bar{b} | \bar{a}$ then $\bar{a}/\bar{b} = \mathsf{fld}^{-1}(\mathsf{fld}(\bar{a}) \cdot \mathsf{fld}(\bar{b})^{-1})$.

A fixed-point rational number $\tilde{x} \in \mathbb{Q}_{\langle k,f \rangle}$ is encoded as an integer $\bar{x} = \tilde{x}2^f \in \mathbb{Z}_{\langle k \rangle}$ and mapped to $\mathbb{Z}_q$ as described above; f and k are public parameters. Secure fixed-point multiplication and division require $q > 2^{2k}$ [6].

Encoding all data types in the same field $\mathbb{Z}_q$ avoids share conversions and thus simplifies the protocols. However, for larger q the running time can be reduced by encoding binary values in small fields.

Notation. We distinguish different representations of a number as follows: we denote $\tilde{x}$ a fixed-point rational number, $\bar{x}$ the integer value of its fixed-point representation, x the field element that encodes $\bar{x}$ (and hence $\tilde{x}$), $[x]$ a sharing of x, and $[x]_i$ the share of party P_i. The notation $x = (\texttt{condition})?\ a : b$ means that the variable x is assigned the value a when `condition`$=true$ and b otherwise.

Table 1. Complexity of core protocols.

Protocol	Rounds	Inv.
$[c]^{\mathbb{F}} \leftarrow [a]^{\mathbb{F}} + [b]^{\mathbb{F}}$	0	0
$[c]^{\mathbb{F}} \leftarrow [a]^{\mathbb{F}}[b]^{\mathbb{F}}$	1	1
$[c]^{\mathbb{F}} \leftarrow [a]^{\mathbb{F}}b$; $[c]^{\mathbb{F}} \leftarrow [a]^{\mathbb{F}} + b$	0	0
$a \leftarrow \mathsf{Output}([a]^{\mathbb{F}})$	1	1
$[z] \leftarrow \mathsf{Inner}([X]^{\mathbb{F}}, [Y]^{\mathbb{F}})$	1	1

Protocol	Rounds	Inv.
$[r] \leftarrow \mathsf{PRandFld}(\mathbb{F})$	0	0
$[r] \leftarrow \mathsf{PRandInt}(k)$	0	0
$[r] \leftarrow \mathsf{PRandBit}()$	1	1 $\mathbb{Z}_q$
$[r] \leftarrow \mathsf{PRandBitL}()$	2	2 $\mathbb{Z}_{q_1}$
$c \leftarrow \mathsf{MulPub}([a], [b])$	1	1 $\mathbb{Z}_q$

Protocol	Rounds	Inv.
$[r'']$, $[r']$, $[r'_{m-1}], \ldots, [r'_0] \leftarrow \mathsf{PRandM}()$	1	m $\mathbb{Z}_q$
$[r]^{\mathbb{F}_{2^8}}, [r] \leftarrow \mathsf{PRandBitD}()$	2	2 $\mathbb{Z}_{q_1}$
$[b]^{\mathbb{F}_{2^8}} \leftarrow \mathsf{BitZQtoF256}([b]^{\mathbb{Z}_q})$	1	1 $\mathbb{Z}_q$
$[b]^{\mathbb{Z}_q} \leftarrow \mathsf{BitF256ToZQ}([b]^{\mathbb{F}_{2^8}}, [r]^{\mathbb{F}_{2^8}}, [r]^{\mathbb{Z}_q})$	1	1 $\mathbb{F}_{2^8}$

Secret random values. Suppose that the parties want to evaluate a function with secret input $[x]$. The task can often be achieved more efficiently using the following technique. The parties jointly generate a shared random value $[r]$, compute $[y] = [x] + [r]$ or $[y] = [x][r]$, and reveal y. Then, they evaluate the function using $[x]$, $[r]$, and y. We obtain $\Delta(x+r, r) = 0$ for $x \in \mathbb{F}$ and uniformly random $r \in \mathbb{F}$ and $\Delta(xr, r) = 0$ for $x \in \mathbb{F}\backslash\{0\}$ and uniformly random $r \in \mathbb{F}\backslash\{0\}$, hence perfect privacy. This is similar to one-time pad encryption of x with key r. Alternatively, for $x \in [0..2^k - 1]$ and random uniform $r \in [0..2^{k+\kappa} - 1]$ we obtain $\Delta(x + r, r) < 2^{-\kappa}$, hence statistical privacy with security parameter κ (only for addition). In this variant, taking $q > 2^{k+\kappa}$ avoids wraparound modulo q when computing $[x] + [r]$ and simplifies certain protocols; the efficiency gain is important (e.g., by eliminating a secure comparison) despite the larger shares.

A shared random integer $r \in [0..2^{k+\kappa}-1]$ with uniform distribution is usually obtained by generating $k + \kappa$ shared random bits $b_0, \ldots, b_{k+\kappa-1}$ and computing $r = \sum_{i=0}^{k+\kappa-1} 2^i b_i$. However, statistical privacy is also achieved for distributions that can be computed more efficiently and/or have particular properties: (1) $r = \sum_i r_i$, for uniformly random $r_i \in [0..2^{k+\kappa} - 1]$; (2) $r = r' + 2^m r''$, where $r' = \sum_{i=0}^{m-1} 2^i b_i$ and $r'' = \sum_i r_i$, for uniformly random $b_i \in \{0, 1\}$ and $r_i \in [0..2^{k+\kappa-m} - 1]$ (Annex A, Theorems 1, 2). We use the second construction for protocols that need $[r'] = [r \bmod 2^m]$ and/or the binary representation of $[r']$.

Shared random values can be generated without interaction using Pseudo-random Replicated Secret Sharing (PRSS) [7] and its integer variant RISS [13]. We define several protocols that use these techniques. $\mathsf{PRandFld}(\mathbb{F})$ generates a uniformly random element of field $\mathbb{F}$ and $\mathsf{PRandInt}(k)$ a random integer $r = \sum_{i=1}^{N} r_i$, for uniformly random $r_i \in [0..2^{k+\kappa} - 1]$ and $N = \binom{n}{t}$. $\mathsf{PRandInt}$ requires a slightly larger modulus, $q > 2^{k+\kappa+\nu}$, $\nu = \lceil \log(N) \rceil$. $\mathsf{MulPub}([a], [b])$ computes the product of two shared field elements with public output as follows: the parties generate a pseudo-random sharing of zero (PRZS [7]) for a polynomial of degree $2t$; each party P_i computes a randomized product of shares $[c]_i = [a]_i[b]_i + [0]_i$; then they exchange the shares and reconstruct $c = ab$. PRSS reduces the com-

plexity of these protocols by 1 round and 1 invocation. Protocol 2.1, PRandBit, returns a shared random bit encoded in $\mathbb{Z}_q$. It combines the protocol RandBit in [11] and PRSS. Finally, Protocol 2.2, PRandM, generates the shared random integers $[r']$ and $[r'']$ defined above, together with the shared bits of r'.

Protocol 2.1: $[b] \leftarrow \mathsf{PRandBit}()$

1 $[r] \leftarrow \mathsf{PRandFld}(\mathbb{Z}_q)$;
2 $u \leftarrow \mathsf{MulPub}([r], [r])$; `// 1 rnd, 1 inv; repeat if` $u = 0$, `pr` $= \frac{1}{q}$
3 $v \leftarrow u^{-(q+1)/4} \bmod q$; `// square root; requires` $q \bmod 4 = 3$
4 $[b] \leftarrow (v[r] + 1)(2^{-1} \bmod q)$;
5 **return** $[b]$;

Protocol 2.2: $([r''], [r'], [b_{m-1}], \ldots, [b_0]) \leftarrow \mathsf{PRandM}(k, m)$

1 $[r''] \leftarrow \mathsf{PRandInt}(k + \kappa - m)$;
2 **foreach** $i \in [0..m-1]$ **do** $[b_i] \leftarrow \mathsf{PRandBit}()$; `// 1 rnd,` m `inv`
3 $[r'] \leftarrow \sum_{i=0}^{m-1} 2^i [b_i]$;
4 **return** $[r''], [r'], [b_{m-1}], \ldots, [b_0]$;

Table 1 lists several protocols for generating shared random bits encoded in small fields, $\mathbb{F}_{2^8}$ and $\mathbb{Z}_{q_1}$, $q_1 > 2^{\kappa+\nu}$, and for bit-share conversions using RISS [14]. PRandBitL generates a random bit shared in $\mathbb{Z}_{q_1}$ and then converts its shares to the target field $\mathbb{Z}_q$. PRandBitD uses a similar technique to generate a random bit shared in both $\mathbb{Z}_q$ and $\mathbb{F}_{2^8}$. Bit-shares in $\mathbb{Z}_q$ are used to construct a random uniform integer, while bit-shares in $\mathbb{F}_{2^8}$ are used for binary computation.

Experiments with an implementation of the protocols [22] showed that these techniques reduce significantly the running time for computation with large integers (or fixed-point numbers): PRandM allows to generate a minimum number of shared random bits, while bit encoding in small fields reduces the communication complexity (smaller shares) as well as the computation complexity (the exponentiation in PRandBit becomes expensive for $\log(q) \approx \log(u) > 256$ bits).

We evaluate the complexity of the protocols assuming that all shared random values are precomputed in parallel using the protocols listed in Table 1. Note that the complexity of PRSS grows quite fast with n. A scalable solution is to run the protocols on a small number of semi-trusted servers [7,12]. Our protocols use PRSS only as an optimization for generating shared random values.

Inner product protocol. Given two shared vectors $[X]$ and $[Y]$, $X, Y \in \mathbb{F}^m$, the obvious method for computing the inner product, $[z] = \sum_{i=1}^{m} [X(i)][Y(i)]$, requires m secure multiplications. We present a more efficient method, that reduces the communication complexity to a single invocation. Assume Shamir sharing for n parties with threshold $t < n/2$. Denote $[X(i)]_j$, $[Y(i)]_j$, $i \in [1..m]$, the input shares and $[z]_j$ the output share of party P_j. The protocol, called Inner, proceeds as follows:

1. Party P_j, $j \in [1..n]$, computes $d_j = \sum_{i=1}^{m} ([X(i)]_j [Y(i)]_j)$ and then shares d_j sending $[d_j]_k$ to party P_k, $k \in [1..n]$.

2. Party P_k, $k \in [1..n]$, computes the share $[z]_k = \sum_{j \in J} \lambda_j [d_j]_k$, where $J \subseteq [1..n]$, $|J| = 2t+1$, and $\{\lambda_j\}_{j \in J}$ is the reconstruction vector for J.

The protocol is a generalization of the secure multiplication of Shamir-shared field elements [16]. The proofs of correctness and security are similar.

Security. For a passive adversary that can corrupt $t < n/2$ parties, the protocols presented in the following can leak information only in steps where they reconstruct shared values. These values are of the form $y = x + r$, where $x \in [0..2^k - 1]$ is the secret and r is a random value constructed using PRandM as described above. It follows from Theorem 2 (Annex A) that $\Delta(y, r) < 2^{-\kappa}$. Since the sub-protocols provide perfect privacy or statistical privacy with the security parameter κ, we conclude that our protocols provide statistical privacy.

Security against an active adversary that can corrupt $t < n/3$ parties can be achieved using several known methods. A variant of the protocols without PRSS can use any extension of Shamir's secret sharing to verifiable secret sharing (VSS) and an associated multiplication proof (e.g., [16,1,8]). An efficient solution for a variant that uses PRSS can be obtained based on the VSS and multiplication proof in [7,13]. The evaluation of these approaches (in particular, of efficient proofs for the inner product) is the subject of on-going work.

3 Integer Arithmetic and Comparison Protocols

Truncation (division by 2^m). The purpose of the truncation protocols is to compute $\lfloor \bar{a}/2^m \rfloor + u$, where $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$, $m \in [1..k-1]$, and $u \in \{0, 1\}$. The bit u depends on the rounding method: $u = 0$ for rounding down, $u = 1$ for rounding up, and $u = (\bar{a}/2^m - \lfloor \bar{a}/2^m \rfloor \geq 0.5)?\ 1 : 0$ for rounding to the nearest integer. Truncation protocols are core components in secure fixed-point arithmetic. We start by reviewing an efficient protocol introduced in [6].

Protocol 3.1, TruncPR, takes on input a secret-shared signed integer $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$ and a public integer $m \in [1..k-1]$, and returns a sharing of $\bar{d} = \lfloor \bar{a}/2^m \rfloor + u$, where $u \in \{0, 1\}$ is a random bit. The protocol rounds to the nearest integer with probability $1 - \alpha$, where α is the distance between $\bar{a}/2^m$ and that integer.

Protocol 3.1: $[d] \leftarrow \mathsf{TruncPR}([a], k, m)$

1 $([r''], [r'], [r'_{m-1}], \ldots, [r'_0]) \leftarrow \mathsf{PRandM}(k, m)$; `// 1 rnd, m inv`
2 $c \leftarrow \mathsf{Output}(2^{k-1} + [a] + 2^m[r''] + [r'])$; `// 1 rnd, 1 inv`
3 $c' \leftarrow c \bmod 2^m$;
4 $[d] \leftarrow ([a] - c' + [r'])(2^{-m} \bmod q)$;
5 **return** $[d]$;

Correctness: Recall that $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$ is encoded in $\mathbb{Z}_q$ as $a = \bar{a} \bmod q$. Let $b = (2^{k-1} + \bar{a}) \bmod q$, $b' = b \bmod 2^m$, and $a' = \bar{a} \bmod 2^m$. Observe that $b \in [0..2^k - 1]$ and $b' = a'$ for any $0 < m < k$. The protocol generates a random secret $r = 2^m r'' + r'$, $r \in [0..2^{k+\kappa+\nu} - 1]$ as explained in Section 2, reveals

$c = (b + r) \bmod q = b + r$, and then computes $c' = c \bmod 2^m$. From $c' = (b' + r') \bmod 2^m$ it follows that $c' - r' = b' - 2^m u = a' - 2^m u$, where $u \in \{0, 1\}$. Step 4 computes $d' = (\bar{a} - a' + 2^m u) \bmod q = (2^m \lfloor \bar{a}/2^m \rfloor + 2^m u) \bmod q$ and then $d = d'(2^{-m} \bmod q) \bmod q = (\lfloor \bar{a}/2^m \rfloor + u) \bmod q$, hence d encodes $\lfloor \bar{a}/2^m \rfloor + u$. Observe that $Pr(u = 1) = Pr(r' + a' \geq 2^m)$, which implies the rounding property.

We extend the method used by TruncPR in order to compute $\bar{a} \bmod 2^m$ and $\lfloor \bar{a}/2^m \rfloor$. The additional task is to determine u. Observe that $u = (c' < r')?1 : 0$ and can be computed using bitwise comparison. This task is achieved by the protocol BitLT (Section 4). We split the computation in two parts: Protocol 3.2, Mod2m, computes $\bar{a} \bmod 2^m$ and Protocol 3.3, Trunc, computes $\lfloor \bar{a}/2^m \rfloor$.

Protocol 3.4, Mod2, handles the case $m = 1$ (extracts the least significant bit). Mod2 is an essential component of the protocols presented in Section 4.

Protocol 3.2: $[a'] \leftarrow \mathsf{Mod2m}([a], k, m)$

1 $([r''], [r'], [r'_{m-1}], \ldots, [r'_0]) \leftarrow \mathsf{PRandM}(k, m)$; // 1 rnd, m inv
2 $c \leftarrow \mathsf{Output}(2^{k-1} + [a] + 2^m[r''] + [r'])$; // 1 rnd, 1 inv
3 $c' \leftarrow c \bmod 2^m$;
4 $[u] \leftarrow \mathsf{BitLT}(c', ([r'_{m-1}], \ldots, [r'_0]))$; // Tables 3, 4
5 $[a'] \leftarrow c' - [r'] + 2^m[u]$;
6 **return** $[a']$;

Protocol 3.3: $[d] \leftarrow \mathsf{Trunc}([a], k, m)$

1 $[a'] \leftarrow \mathsf{Mod2m}([a], k, m)$; // Table 5
2 $[d] \leftarrow ([a] - [a'])(2^{-m} \bmod q)$;
3 **return** $[d]$;

Protocol 3.4: $[a_0] \leftarrow \mathsf{Mod2}([a], k)$

1 $([r''], [r'], [r'_0]) \leftarrow \mathsf{PRandM}(k, 1)$; // 1 rnd, 1 inv
2 $c \leftarrow \mathsf{Output}(2^{k-1} + [a] + 2[r''] + [r'_0])$; // 1 rnd, 1 inv
3 $[a_0] \leftarrow c_0 + [r'_0] - 2c_0[r'_0]$;
4 **return** $[a_0]$;

Truncation with deterministic rounding to the nearest integer can be obtained by adding to Protocol 3.3 the following steps: $[v] \leftarrow \mathsf{LTZ}([a'] - 2^{m-1}, m)$ and $[e] \leftarrow [d] + 1 - [v]$, where LTZ computes $(a' < 2^{m-1})?\ 1 : 0$. However, this solution is much more complex than TruncPR.

Integer division with public divisor. Truncation can be generalized in order to obtain protocols that on input a shared $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$ and a public $x \in [1..2^{k-1} - 1]$ compute shares of the quotient $\lfloor \bar{a}/x \rfloor$ and the remainder $\bar{a} \bmod x$. Division can better be handled by secure fixed-point arithmetic [6]. However, integer division with public divisor is relatively simple, and sufficient for some applications [17].

One approach is to adapt Mod2m by replacing 2^m with x. This generalization was first observed in [17]. However, generating a secret r' with uniform distri-

bution in $[0..x-1]$ instead of $[0..2^m-1]$ is more complex. Let $m = \lceil \log(x) \rceil$. The method used in [17] generates $r' \in [0..2^m-1]$ from random bits and tests if $r' < x$. This method succeeds after $2^m/x < 2$ iterations on average, but the iterations include secure comparisons and are expensive. Protocol 3.5, Mod, avoids the iterative search for r' at the cost of replacing a bitwise comparison by an integer comparison. The protocol for computing the quotient is similar to Trunc.

Correctness: Let $b = (2^{k-1} + a) \bmod q$ and $b' = b \bmod x = \bar{a} \bmod x$. The protocol generates the secret random $r = xr'' + r'$ with $r' \in [0..2^m-1]$, reveals $c = b + r$, and computes $c' = c \bmod x$. Observe that $r \bmod x = r' \bmod x = r' - xv$, where $v = (r' \geq x)?1 : 0$. Furthermore, $c' = (b' + (r' \bmod x)) \bmod x = (b' + r' - xv) \bmod x$, hence $b' = c' - r' + xv + xu$, where $u = (c' < r' - xv)?1 : 0$. Choosing $r' \in [0..2^m-1]$ instead of $r' \in [0..x-1]$ preserves statistical privacy.

Protocol 3.5: $[a'] \leftarrow \mathsf{Mod}([a], k, x)$

1 $m \leftarrow \lceil \log(x) \rceil$;
2 $([r''], [r'], [r'_{m-1}], \ldots, [r'_0]) \leftarrow \mathsf{PRandM}(k, m)$; `// 1 rnd, m inv`
3 $c \leftarrow \mathsf{Output}(2^{k-1} + [a] + x \cdot [r''] + [r'])$; `// 1 rnd, 1 inv`
4 $c' \leftarrow c \bmod x$;
5 $[v] \leftarrow 1 - \mathsf{BitLT}(([r'_{m-1}], \ldots, [r'_0]), x)$; `// Tables 3, 4`
6 $[u] \leftarrow \mathsf{LTZ}(c' - [r'] + x[v], m)$; `// Table 5`
7 $[a'] \leftarrow c' - [r'] + x([v] + [u])$;
8 **return** $[a']$;

Inequality and equality. The family of secure integer comparison operators with secret inputs and outputs (Table 2) can be constructed based on two primitives: Protocol 3.6, $\mathsf{LTZ}([a], k)$, that computes $(\bar{a} < 0)?1 : 0$, and Protocol 3.7, $\mathsf{EQZ}([a], k)$, that computes $(\bar{a} = 0)?1 : 0$, for $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$.

Table 2. Protocols for integer comparison.

Op.	Protocol	Construction	Op.	Protocol	Construction
$a = 0$	$\mathsf{EQZ}(a)$	Primitive	$a = b$	$\mathsf{EQ}(a,b)$	$\mathsf{EQ}(a,b) = \mathsf{EQZ}(a-b)$
$a < 0$	$\mathsf{LTZ}(a)$	Primitive (sign of a)	$a < b$	$\mathsf{LT}(a,b)$	$\mathsf{LT}(a,b) = \mathsf{LTZ}(a-b)$
$a > 0$	$\mathsf{GTZ}(a)$	$\mathsf{GTZ}(a) = \mathsf{LTZ}(-a)$	$a > b$	$\mathsf{GT}(a,b)$	$\mathsf{GT}(a,b) = \mathsf{LTZ}(b-a)$
$a \leq 0$	$\mathsf{LEZ}(a)$	$\mathsf{LEZ}(a) = 1 - \mathsf{LTZ}(-a)$	$a \leq b$	$\mathsf{LE}(a,b)$	$\mathsf{LE}(a,b) = 1 - \mathsf{LTZ}(b-a)$
$a \geq 0$	$\mathsf{GEZ}(a)$	$\mathsf{GEZ}(a) = 1 - \mathsf{LTZ}(a)$	$a \geq b$	$\mathsf{GE}(a,b)$	$\mathsf{GE}(a,b) = 1 - \mathsf{LTZ}(a-b)$

LTZ is based on the following remark: if $\bar{a} < 0$ then $\lfloor \bar{a}/2^{k-1} \rfloor = -1$ and if $\bar{a} \geq 0$ then $\lfloor \bar{a}/2^{k-1} \rfloor = 0$. Therefore, we can determine the sign of a secret integer $\bar{a} \in \mathbb{Z}_{\langle k \rangle}$ by computing $[s] = -\mathsf{Trunc}([a], k, k-1)$.

EQZ starts by computing and revealing $c = \bar{a} + r$ like the previous protocols. Let $c' = c \bmod 2^k$ and observe that $c' = r'$ if and only if $\bar{a} = 0$. The protocol computes $z = (c' = r')? \ 1 : 0 = \bigvee_{i=0}^{k-1} (c_i \oplus r'_i)$ using the k-ary OR protocol KOr (Section 4). EQZ is similar to a protocol in [19].

Protocol 3.6: $[s] \leftarrow \mathsf{LTZ}([a], k)$

1 $[s] \leftarrow -\mathsf{Trunc}([a], k, k-1)$; // Table 5
2 **return** $[s]$;

Protocol 3.7: $[z] \leftarrow \mathsf{EQZ}([a], k)$

1 $([r''], [r'], [r'_{k-1}], \ldots, [r'_0]) \leftarrow \mathsf{PRandM}(k, k)$; // 1 rnd, k inv
2 $c \leftarrow \mathsf{Output}([a] + 2^k[r''] + [r'])$; // 1 rnd, 1 inv
3 $(c_{k-1}, \ldots, c_0) \leftarrow Bits(c, k)$;
4 **foreach** $i \in [0..k-1]$ **do** $[d_i] \leftarrow c_i + [r'_i] - 2c_i[r'_i]$;
5 $[z] \leftarrow 1 - \mathsf{KOr}([d_{k-1}], \ldots, [d_0])$; // Tables 3, 4
6 **return** $[z]$;

4 Building Blocks

The performance of the protocols presented in the previous section is determined by the complexity of the building blocks for bitwise operations. We discuss in this section protocols for different trade-offs between communication and round complexity, adapted to different types of application and execution environment.

Let $\mathcal{A}$ be a set and $\odot : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ an associative binary operator. We consider the following two extensions: a k-ary operation computes $p = \bigodot_{i=1}^{k} a_i$; a prefix operation computes $p_j = \bigodot_{i=1}^{j} a_i$ for $j \in [1..k]$. We focus here on multiplication and simple boolean functions, especially OR and carry propagation.

4.1 Bitwise Operations with Logarithmic Round Complexity

For secure computation with large batches of parallel operations (e.g., large numbers and/or large matrices in linear programming), we can improve the performance by trading off a few rounds for low communication overhead.

Protocols with perfect privacy, low communication, and $\log(k)$ rounds can be obtained by structuring the computation of k-ary and prefix operations as shown in Fig. 1. The protocols for k-ary operations have minimum communication complexity. For prefix operations, the decision is less obvious, since optimal communication complexity requires $2\log(k) - 1$ rounds for $2k - \log(k) - 2$ invocations. The structure in Fig. 1 offers a better trade-off.

Complexity is shown in Table 3, where KOpL and PreOpL are generic protocols for any binary operation computed in 1 invocation, while CarryOutL and CarryAddL compute the carry bits for binary addition (prefix operation, details in [22]). Note that these protocols work for any data encoding (Section 2), in particular for bits encoded in small fields.

Protocol 4.1, BitLTL, is an efficient inequality test in $\log(k)$ rounds for bitwise encoded integers $a, b \in [0..2^k - 1]$. The protocol computes $s = (a < b)? \ 1 : 0$ with perfect privacy. We show the variant used in Section 3, where one integer

Table 3. Complexity of log-rounds bitwise operations.

Protocol	Rounds	Invocations ($\mathbb{F}$)
$[p] \leftarrow \mathsf{KOpL}([a_1], \ldots, [a_k])$	$\log(k)$	$k-1$
$[p_1], \ldots, [p_k] \leftarrow \mathsf{PreOpL}([a_1], \ldots, [a_k])$	$\log(k)$	$0.5k\log(k)$
$[c] \leftarrow \mathsf{CarryOutL}(a_k, \ldots, a_1, [b_k], \ldots, [b_1], c')$	$\log(k)$	$2k-2$
$[c_k], \ldots, [c_1] \leftarrow \mathsf{CarryAddL}(a_k, \ldots, a_1, [b_k], \ldots, [b_1])$	$\log(k)$	$k\log(k)$
$[c] \leftarrow \mathsf{CarryOutL}([a_k], \ldots, [a_1], [b_k], \ldots, [b_1], c')$	$\log(k)+1$	$3k-2$
$[c_k], \ldots, [c_1] \leftarrow \mathsf{CarryAddL}([a_k], \ldots, [a_1], [b_k], \ldots, [b_1])$	$\log(k)+1$	$k\log(k)+k$
$[s] \leftarrow \mathsf{BitLTL}(a, [b_k], \ldots, [b_1])$	$\log(k)$	$2k-2$

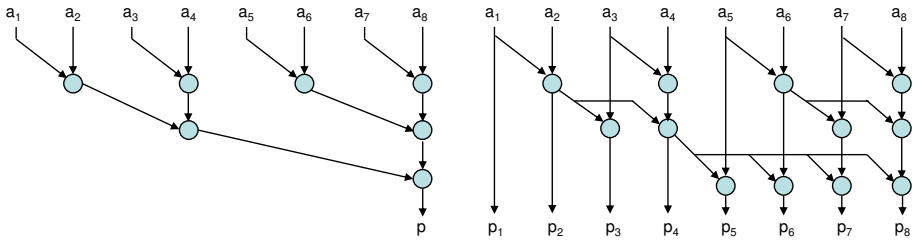

Fig. 1. K-ary (left) and prefix (right) operations in $\log(k)$ rounds (for $k = 8$).

is public. A solution using PreOrL and the algorithm in [11] is less efficient, since it needs $0.5k\log(k)$ invocations (or $2\log(k)$ rounds and $2k - \log(k)$ invocations).

Correctness: Let $d = 2^k + a - b$ and $d_{k+1}, \ldots, d_1$ its binary representation. Observe that $0 < d < 2^{k+1}$ and $d_{k+1} = (a - b < 0)? \, 0 : 1$, hence $s = 1 - d_{k+1}$. On the other hand, $2^k - b = b' + 1$, where $b' = \sum_{i=1}^{k} 2^{i-1} \neg b_i$, and hence $d = a + b' + 1$. The protocol computes d_{k+1} using $\mathsf{CarryOutL}$, for inputs a, b', and carry-in set.

Protocol 4.1: $[s]^{\mathbb{F}} \leftarrow \mathsf{BitLTL}(a, [b_k]^{\mathbb{F}}, \ldots, [b_1]^{\mathbb{F}})$

1 $(a_k, \ldots, a_1) \leftarrow Bits(a, k)$;
2 **foreach** $i \in [1..k]$ **do** $[b'_i]^{\mathbb{F}} \leftarrow 1 - [b_i]^{\mathbb{F}}$;
3 $[s]^{\mathbb{F}} \leftarrow 1 - \mathsf{CarryOutL}(a_k, \ldots, a_1, [b'_k]^{\mathbb{F}}, \ldots, [b'_1]^{\mathbb{F}}, 1)$; `// Table 3`
4 **return** $[s]^{\mathbb{F}}$;

4.2 Bitwise Operations with Constant Round Complexity

For applications with many sequential operations, the running time is dominated by the network delay (e.g., secure division by functional iteration). It is important, therefore, to reduce the number of rounds of these operations, even at the cost of larger amount of data. Furthermore, precomputation becomes an effective method for reducing the running time, so another design goal is to shift complexity to a precomputation phase. We present new protocols that meet these goals, as well as methods to reduce the complexity of known protocols. Table 4

Table 4. Complexity of constant-rounds bitwise operations.

Protocol	Rounds	Invocations ($\mathbb{Z}_q$)
$[p_1],\ldots,[p_k] \leftarrow \mathsf{PreMulC}([a_1],\ldots,[a_k])$	2	$3k-1$
After precomputation	1	k
$[p_1],\ldots,[p_k] \leftarrow \mathsf{PreOrC}([a_1],\ldots,[a_k])$	3	$5k-1$
After precomputation	2	$2k-1$
$[p] \leftarrow \mathsf{KOrCL}([a_1],\ldots,[a_k])$	3	$4\log(k)$
After precomputation	2	$\log(k)+1$
$[p_1],\ldots,[p_k] \leftarrow \mathsf{PreOrCS}([a_1],\ldots,[a_k])$	6	$8k+3\sqrt{k}$
After precomputation	5	$4k+\sqrt{k}$
$[p] \leftarrow \mathsf{KOrCS}([a_1],\ldots,[a_k])$	2	$3k-1$
After precomputation	1	k
$\mathsf{BitLTC1}(a,[b_k],\ldots,[b_1])$	3	$3k+1$
After precomputation	2	$k+1$
$\mathsf{BitLTC2}(a,[b_k],\ldots,[b_1])$	5	$2k-1$
After precomputation	4	$k+1$

summarizes the complexity of the protocols discussed in this section. We show the protocols for k-ary and prefix-OR; variants for AND can be obtained using De Morgan's laws. We refer to several known protocols as follows: PreMulC, for prefix multiplication [4]; KOrCS and PreOrCS, for k-ary and prefix OR [11].

We begin with an important building block, PreMulC, that computes in constant-rounds the prefix products $[p_j] = \prod_{i=1}^{j}[a_i]$, $j \in [1..k]$, where $a_1,\ldots,a_k$ are *non-zero* elements of a field $\mathbb{F}$. The protocol follows the method introduced in [4]: the parties compute $[m_i] = [r_i][a_i][r_{i-1}^{-1}]$, $i \in [2..k]$ and $[m_1] = [r_1][a_1]$, where r_i are uniformly random in $\mathbb{F}\backslash\{0\}$; reveal m_i; and then locally compute the prefix products $[p_j] = [r_j^{-1}]\prod_{i=1}^{j} m_i = [r_j^{-1}]r_j a_j r_{j-1}^{-1}\ldots r_2 a_2 r_1^{-1} r_1 a_1$.

Protocol 4.2: $[p_1],\ldots,[p_k] \leftarrow \mathsf{PreMulC}([a_1],\ldots,[a_k])$

foreach $i \in [1..k]$ **do parallel**
  $[r_i] \leftarrow \mathsf{PRandFld}(\mathbb{F})$;
  $[s_i] \leftarrow \mathsf{PRandFld}(\mathbb{F})$;
  $u_i \leftarrow \mathsf{MulPub}([r_i],[s_i])$; `// 1 rnd, k inv; repeat if` $u_i = 0$, `pr` $= \frac{2}{|\mathbb{F}|}$
foreach $i \in [1..k-1]$ **do parallel** $[v_i] \leftarrow [r_{i+1}][s_i]$; `//` $k-1$ `inv`
$[w_1] \leftarrow [r_1]$;
foreach $i \in [2..k]$ **do** $[w_i] \leftarrow [v_{i-1}](u_{i-1}^{-1} \bmod q)$;
foreach $i \in [1..k]$ **do** $[z_i] \leftarrow [s_i](u_i^{-1} \bmod q)$;
foreach $i \in [1..k]$ **do parallel** $m_i \leftarrow \mathsf{MulPub}([w_i][a_i])$; `// 1 rnd,` k `inv`
$[p_1] \leftarrow [a_1]$;
foreach $j \in [2..k]$ **do** $[p_j] \leftarrow [z_j](\prod_{i=1}^{j} m_i)$;
return $([p_1],\ldots,[p_k])$;

The variant of prefix multiplication shown as Protocol 4.2 minimizes the number of rounds as follows: steps 1-8 (round 1) compute $[w_1] = [r_1 a_1]$, $[w_i] = [r_i][r_{i-1}]^{-1}$, $i \in [2..k]$, and $[z_i] = [r_i^{-1}]$, $i \in [1..k]$; steps 9-11 (round 2) complete the computation described above. The protocol gains 1 round with respect to the obvious solution by computing in parallel $r_i s_i$ and $r_{i+1} s_i$. The first round can be pre-computed.

The complexity of all the other protocols listed in Table 4 is evaluated assuming that prefix multiplication is optimized as shown in Protocol 4.2, and shared random values are pre-computed in parallel.

The protocol for k-ary symmetric boolean functions proposed in [11] has the same complexity as PreMulC. We can reduce the complexity of k-ary OR (and k-ary AND) from $3k$ to $4\log(k)$ invocations as shown in Protocol 4.3, KOrCL. In particular, this reduces the complexity of the equality test EQZ. KOrCL is based on the remark that $\bigvee_{i=1}^{k} a_i = 0$ if and only if $\sum_{i=1}^{k} a_i = 0$, which can be tested more efficiently (like in EQZ), since $\sum_{i=1}^{k} a_i \leq k$.

Protocol 4.3: $[e] \leftarrow \mathsf{KOrCL}([a_1], \ldots, [a_k])$

1 $m \leftarrow \lceil \log(k) \rceil$;
2 $([r''], [r'], [r'_{m-1}], \ldots, [r'_0]) \leftarrow \mathsf{PRandM}(k, m)$; // 1 rnd, m inv
3 $c \leftarrow \mathsf{Output}(2^m [r''] + [r'] + \sum_{i=1}^{k} [a_i])$; // 1 rnd, 1 inv
4 $(c_m, \ldots, c_1) \leftarrow Bits(c, m)$;
5 **foreach** $i \in [1..m]$ **do** $[d_i] \leftarrow c_i + [r_i] - 2c_i[r'_i]$;
6 $[e] \leftarrow \mathsf{KOrCS}([d_1], \ldots, [d_m])$; // 2 rnd, $3m - 1$ inv
7 **return** $[e]$;

Protocol 4.4, PreOrC, computes prefix-OR in 3 rounds, out of which 1 round can be precomputed. PreOrC is more efficient (and much simpler) than the prefix-OR protocol described in [11] (PreOrCS in Table 4).

Correctness: PreOrC computes the prefix products $b_i = \prod_{j=1}^{i} (a_j + 1)$ for $i \in [1..k]$. Observe that if $a_j = 0$ for $j \in [1..i]$ then $b_i = 1$, otherwise b_i is even. Therefore, the complement of the least significant bit of b_i is equal to $\bigvee_{j=1}^{i} a_i$.

Protocol 4.4: $[p_1], \ldots, [p_k] \leftarrow \mathsf{PreOrC}([a_1], \ldots, [a_k])$

1 $[b_1], \ldots, [b_k] \leftarrow \mathsf{PreMulC}([a_1] + 1, \ldots, [a_k] + 1)$; // 2 rnd, $3k - 1$ inv
2 $[p_1] \leftarrow [a_1]$;
3 **foreach** $i \in [2..k]$ **do parallel**
4 $\quad [p_i] \leftarrow 1 - \mathsf{Mod2}([b_i], k)$; // 2 rnd, $2k - 2$ inv
5 **return** $[p_1], \ldots, [p_k]$;

We conclude this section with two constant-rounds variants of the protocol for comparison of bitwise encoded integers. The protocol could be constructed using PreOrC and the method in [11]. However, we can eliminate $2k$ invocations as shown in Protocol 4.5, BitLTC1. This algorithm is also more efficient than the transformation proposed in [20], saving 1 round and k invocations.

Protocol 4.5: $[u] \leftarrow \mathsf{BitLTC1}(a, ([b_k], \ldots, [b_1]))$

1 **foreach** $i \in [1..k]$ **do** $[d_i] \leftarrow a_i + [b_i] - 2a_i[b_i]$;
2 $([p_k], \ldots, [p_1]) \leftarrow \mathsf{PreMulC}([d_k] + 1, \ldots, [d_1] + 1)$; `// 2 rnd, 3k − 1 inv`
3 **foreach** $i \in [1..k-1]$ **do** $[s_i] \leftarrow [p_i] - [p_{i+1}]$;
4 $[s_k] \leftarrow [p_k] - 1$;
5 $[s] \leftarrow \sum_{i=1}^{k} [s_i](1 - a_i)$;
6 $[u] \leftarrow \mathsf{Mod2}([s], k)$; `// 2 rnd, 2 inv`
7 **return** $[u]$;

Correctness: Step 1 determines the bit differences $d_i = a_i \oplus b_i$ and step 2 computes the prefix products $p_i = \prod_{j=i}^{k}(d_j + 1) = 2^{\sum_{j=i}^{k} d_j}$ for $i \in [1..k]$. Steps 3-4 compute $s_i = p_i - p_{i+1} = d_i p_{i+1}$ for $i \in [1..k-1]$ and $s_k = p_k - 1 = d_k$. Steps 5-6 compute $s = \sum_{i=1}^{k} s_i(1 - a_i) = d_k(1 - a_k) + \sum_{i=1}^{k-1} s_i(1 - a_i)$ and extract the least significant bit of s, denoted u. We distinguish the following cases:

- If $a = b$ then $d_i = 0$, $p_i = 1$, and $s_i = 0$ for $i \in [1..k]$. Therefore, $s = 0$ and hence $u = 0 = (a < b)$. Otherwise, $a \neq b$ and there exists $m \in [1..k]$ such that $a_i = b_i$ for all $i > m$ and $a_m \neq b_m$. Observe that $(a < b) = b_m = 1 - a_m$.
- If $m = k$ then $a_k \neq b_k$, $d_k = 1$, $p_k = 2$, $s_k = 1$, and $s_{k-1} = d_{k-1}p_k = 2d_{k-1}$; for $i \in [1..k-2]$, $s_i = d_i p_{i+1} = d_i 2^{\sum_{j=i+1}^{k} d_j} = 2d_i 2^{\sum_{j=i+1}^{k-1} d_j}$. It follows that $s = (1 - a_k) + 2d_{k-1} + 2\sum_{i=1}^{k-2} d_i(1 - a_i)2^{\sum_{j=i+1}^{k-1} d_j}$, hence $u = 1 - a_k = (a < b)$.
- If $m < k$ then for $i \in [m+1..k]$ we have $a_i = b_i$, $d_i = 0$, $p_i = 1$, and $s_i = p_{i+1}d_i = 0$; if $i = m$ then $d_m = 1$, $p_m = 2$, and $s_m = 1$; for $i \in [1..m-1]$, $s_i = d_i p_{i+1} = 2d_i 2^{\sum_{j=i+1, j\neq m}^{k} d_j}$. It follows that $s = (1 - a_m) + 2\sum_{i=1}^{m-1} d_i(1 - a_i)2^{\sum_{j=i+1, j\neq m}^{k} d_j}$, hence $u = 1 - a_m = (a < b)$.

Protocol 4.6, BitLTC2, is a variant that reduces the communication complexity by splitting the computation of the prefix products in 3 steps, at the cost of 2 additional rounds. We omit the correctness proof which is simple but lengthy.

Protocol 4.6: $[u] \leftarrow \mathsf{BitLTC2}(a, ([b_k], \ldots, [b_1]))$

1 **foreach** $i \in [1..k]$ **do** $[d_i] \leftarrow a_i + [b_i] - 2a_i[b_i]$;
2 $\ell \leftarrow (k-1)/2$; `// Assume that k is odd`
3 **foreach** $i \in [1..\ell]$ **do** $[e_i] \leftarrow ([d_{2i-1}] + 1)([d_{2i}] + 1)$; `// 1 rnd, k/2 inv`
4 $[e_{\ell+1}] \leftarrow [d_k] + 1$;
5 $([f_\ell], \ldots, [f_1]) \leftarrow \mathsf{PreMulC}([e_{\ell+1}], \ldots, [e_2])$; `// 2 rnd, 3k/2 inv`
6 **foreach** $i \in [1..\ell]$ **do** $[g_i] \leftarrow ([e_i] - 1)(1 - a_{2i-1}) + [d_{2i}](a_{2i-1} - a_{2i})$;
7 $[s] \leftarrow \mathsf{Inner}(([f_1], \ldots, [f_\ell]), ([g_1], \ldots, [g_\ell]))$; `// 1 rnd, 1 inv`
8 $[s] \leftarrow [s] + [d_k](1 - a_k)$;
9 $[u] \leftarrow \mathsf{Mod2}([s], k)$; `// 2 rnd, 2 inv`
10 **return** $[u]$;

5 Conclusions

Comparison and truncation are the main performance bottlenecks of applications that involve complex secure computation with integer and fixed-point rational numbers. We focus in this paper on improved solutions for these tasks. We propose efficient protocols for their building blocks, with complexity reduced by a large margin with respect to known protocols. These building blocks are often used in other secure computation tasks. Moreover, we give variants for different trade-offs between communication and round complexity, and thus the protocols can be adapted to applications and the communication network.

Table 5. Complexity of arithmetic and comparison protocols.

Protocol	Rounds	Inv. ($\mathbb{Z}_q$)
Mod2mC1$([a], k, m)$	4	$4m + 1$
After prec.	3	$m + 2$
Mod2mC2$([a], k, m)$	6	$3m + 1$
After prec.	5	$m + 3$
Mod2mL$([a], k, m)$	$\log(m) + 2$	$3m - 1$
After prec.	$\log(m) + 1$	$2m - 1$
TruncPR$([a], k, m)$	2	$m + 1$
After prec.	1	1

Protocol	Rounds	Inv. ($\mathbb{Z}_q$)
LTZC1$([a], k)$	4	$4k - 2$
After prec.	3	$k + 1$
LTZC2$([a], k)$	6	$3k - 2$
After prec.	5	$k + 2$
LTZL$([a], k)$	$\log(k) + 2$	$3k - 4$
After prec.	$\log(k) + 1$	$2k - 3$
EQZC$([a], k)$	4	$k + 4\log(k)$
After prec.	3	$\log(k) + 2$

Protocol	Rounds	Inv.	Field
$[s] \leftarrow$ LTZL$([a], k)$	1	1	$\mathbb{Z}_q$
	2	$2k$	$\mathbb{Z}_{q_1}$
	$\log(k) + 1$	$2k - 3$	$\mathbb{F}_{2^s}$
After precomputation	1	1	$\mathbb{Z}_q$
	$\log(k) + 1$	$2k - 3$	$\mathbb{F}_{2^s}$

Table 5 shows the complexity of the main protocols in Section 3. Mod2mC1, Mod2mC2, and Mod2mL are variants of Mod2m constructed with the BitLT variants BitLTC1, BitLTC2, and BitLTL, respectively. The same naming convention is used for LTZ, while EQZC is a variant of EQZ constructed with KOrCL.

Besides the usual setting with all data types encoded in the same field $\mathbb{Z}_q$, we also show a variant of LTZL with bits encoded in small fields ($\mathbb{F}_{2^s}$ for BitLTL, $\mathbb{Z}_{q_1}$ for random bit generation). This variant reduces the amount of data sent from $O(k^2)$ bits to $O(k)$ and thus scales up better for large integers (or fixed-point numbers) and large batches of comparisons. Our measurements for secure linear programming [22] showed that the running time of LTZL is reduced by a factor of 1.6 for $\log(q) = 128$ and 2.5 for $\log(q) = 320$, $\kappa = 48$. On the other hand, using PreOrC and LTZC1 instead of log-rounds protocols reduces the round complexity of the fixed-point division protocol in [6] by 10 rounds (for signed dividend).

From a practical perspective, the comparison protocols (inequality and equality) listed in Table 5 are more efficient than the protocols in [19,24,20]. We note,

however, that their work focuses on solutions with perfect privacy, while our protocols offer (only) statistical privacy. Reconciliating these goals, performance and perfect privacy, and finding multiparty comparison and (accurate) truncation protocols with sub-linear complexity, remain open and challenging issues.

Acknowledgements. Part of this work was funded by the European Commission through the grant FP7-213531 to the SecureSCM project.

References

1. M. Abe, R. Cramer, and S. Fehr. Non-interactive Distributed-Verifier Proofs and Proving Relations among Commitments. In *ASIACRYPT 2002*, volume 2501 of *LNCS*. Springer, 2002.
2. J. Algesheimer, J. Camenish, and V. Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *CRYPTO 2002*, volume 2442 of *LNCS*, pages 417–432. Springer, 2002.
3. M. Atallah, M. Blanton, V. Deshpande, K. Frikken, J. Li, and L. Schwarz. Secure Collaborative Planning, Forecasting, and Replenishment (SCPFR). In *Multi-Echelon/Public Applications of Supply Chain Management Conference*, 2006.
4. J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds of interaction. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, pages 201–209. ACM, 1989.
5. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault tolerant distributed computation. In *Proc. of 20th ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 1988.
6. O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography and Data Security*, LNCS. Springer, 2010.
7. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proc. of 2nd Theory of Cryptography Conference (TCC'05)*, pages 342–362, 2005.
8. R. Cramer, I. Damgård, and U. Maurer. General Secure Multi-Party Computation from any Linear Secret-Sharing Scheme. In *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 316–334. Springer, 2000.
9. R. Cramer, I. Damgård, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 280–300. Springer, 2001.
10. R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *EUROCRYPT 1997*, volume 1233 of *LNCS*, pages 103–118. Springer, 1997.
11. I. Damgård, M. Fitzi, E. Kiltz, J. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Proc. of 3rd Theory of Cryptography Conference (TCC'06)*, volume 3876 of *LNCS*, pages 285–304. Springer, 2006.
12. I. Damgård, J. Nielsen, T. Toft, J. I. Pagter, T. Jakobsen, P. Bogetoft, and K. Nielsen. A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation. In *Proc. of Financial Cryptography 2006*, volume 4107 of *LNCS*, pages 142–147. Springer, 2006.
13. I. Damgård and R. Thorbek. Non-interactive Proofs for Integer Multiplication. In *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 412–429. Springer, 2007.

14. I. Damgård and R. Thorbek. Efficient conversion of secret-shared values between different fields. Cryptology ePrint Archive, Report 2008/221, 2008.
15. J. Garay, B. Schoenmakers, and J. Villegas. Practical and Secure Solutions for Integer Comparison. In *PKC 2007*, volume 4450 of *LNCS*. Springer, 2007.
16. R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multi-party computations with applications to threshold cryptography. In *Proc. of ACM Symposium on Principles of Distributed Computing (PODC'98)*, 1998.
17. J. Guajardo, B. Mennink, and B. Schoenmakers. Modulo Reduction for Paillier Encryptions and Application to Secure Statistical Analysis. In *Financial Cryptography and Data Security*, LNCS. Springer, 2010.
18. F. Kerschbaum. Building a Privacy-Preserving Benchmarking Enterprise System. *Enterp. Inf. Syst.*, 2(4):421–441, 2008.
19. T. Nishide and K. Ohta. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In *PKC 2007*, volume 4450 of *LNCS*, pages 343–360. Springer, 2007.
20. T. I. Reistad. Multiparty comparison - an improved multiparty protocol for comparison of secret-shared values. In *SECRYPT*, pages 325–330. INSTICC, 2009.
21. B. Schoenmakers and P. Tuyls. Efficient Binary Conversion for Paillier Encryptions. In *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 522–537. Springer, 2006.
22. SecureSCM. Security Analysis. Deliverable D9.2, EU FP7 Project Secure Supply Chain Management (SecureSCM), 2009.
23. V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2nd edition, 2009.
24. T. Toft. *Primitives and Applications for Multi-party Computation*. PhD dissertation, University of Aarhus, Denmark, 2007.
25. T. Toft. Solving Linear Programs Using Multiparty Computation. In *Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 2009.

A Annex: Statistical Privacy

The protocols presented in the paper offer two notions of privacy: *perfect* and *statistical*. Some of the protocols provide only statistical security in order to achieve better efficiency. The difference between these notions is discussed below.

Definition 1. *Let X and Y be two random variables, both taking values in some finite set V. The statistical distance between X and Y is defined as*

$$\Delta(X;Y) = \frac{1}{2}\sum_{v\in V} |\mathbb{P}(X=v) - \mathbb{P}(Y=v)|. \tag{1}$$

Intuitively, if $\Delta(X;Y)$ is small (or 0), then the distributions of X and Y are statistically (resp. perfectly) indistinguishable. All our building blocks are based on the following high-level idea. Let x be a random variable denoting a secret integer. We first generate a random secret integer r in some range and reveal $f(x,r)$ for some function f. Let $\delta = \Delta(r, f(x,r))$. The type of security offered by the protocol depends on δ as follows: $\delta = 0$ implies perfect security and $\delta \leq c/2^{\kappa}$ (for some constant c) implies statistical security in security parameter κ.

Following are some basic results about statistical distance. Their proofs can be found in Chapter 8 of [23].

We first show that if U is uniform on some finite set then the statistical distance between $X+U$ and U can be bounded by the size of the domain of U.

Lemma 1. *Let M and K be positive integers with $M \leq K$. Let X, U be random variables in $[0..M-1]$, $[0..K-1]$ respectively such that U is uniform. Then $\Delta(U; X+U) \leq (M-1)/K$ and this bound is tight.*

Proof. This is Lemma 1 in [21, Appendix A].

Remark 1. The result of Lemma 1 implies that $\Delta(U; X+U)$ is small if $M \ll K$. For instance, if one sets $K = M2^k$, we see that the statistical distance between U and $X+U$ is less than $1/2^k$, hence approaches 0 exponentially fast as a function of k. In other words, one can mask an integer value X from a bounded range $\{0, \ldots, M-1\}$ by adding a uniform random integer U from an enlarged range $\{0, \ldots, K-1\}$. This way one can do one-time pad encryption with integers, where X is the message, U is the one-time pad, and $X+U$ is the ciphertext.

In Theorem 1, we show that this holds even if U is not uniform, but a sum of uniform distributions. For this we will use the following lemmas.

Lemma 2. *Let X and Y be random variable taking values in some finite set V and let $f : V \to V'$ be some function mapping to some finite set V'. It holds that*

$$\Delta(f(X); f(Y)) \leq \Delta(X; Y). \tag{2}$$

Proof. This is Theorem 8.32 of [23].

Lemma 3. *Let X,Y and Z be random values, where X and Z are independent and Y and Z are independent, then*

$$\Delta((X, Z); (Y, Z)) = \Delta(X; Y). \tag{3}$$

Proof. This is Theorem 8.33 of [23].

Theorem 1. *Let $X \in [0..2^k-1]$ and U be random variables and let $U = \sum_{i=1}^{n} U_i$ for some finite n, where each U_i is independent and uniform in $[0..2^{k+\kappa}-1]$. Then:*

$$\Delta(X+U; U) < 2^{-\kappa}. \tag{4}$$

Proof. Let $U_i \in_R [0..2^{k+\kappa}-1]$ for $i = 1, \ldots, n$ such that U_i is selected uniformly and let and $X \in [0..2^k-1]$ be with unknown distribution. Let $U = \sum_{i=1}^{n} U_i$. Lastly, let $f : [0..(n-1)2^{k+\kappa}-n+1] \times [0..2^k(1+2^\kappa)-2] \to [0..2^k(1+n2^\kappa)-n-1]$

be defined as $f(x,y) := x + y$. It follows that

$$\begin{aligned}
\Delta(X+U;U) &= \Delta(X+\sum_{i=1}^{n} U_i; \sum_{i=1}^{n} U_i) = \Delta(X+\sum_{i=1}^{n-1} U_i + U_n; \sum_{i=1}^{n-1} U_i + U_n) \\
&= \Delta(f(\sum_{i=1}^{n-1} U_i, X+U_n); f(\sum_{i=1}^{n-1} U_i, U_n)) \\
&\text{Lemma 2} \leq \Delta((\sum_{i=1}^{n-1} U_i, X+U_n); (\sum_{i=1}^{n-1} U_i, U_n)) \\
&\text{Lemma 3} = \Delta(X+U_n; U_n) \\
&\text{Lemma 1} \leq \frac{2^k-1}{2^{k+\kappa}} < 2^{-\kappa}.
\end{aligned}$$

Theorem 2 is an extension of Theorem 1 where U is constructed in a slightly different manner.

Theorem 2. *Let $X \in [0..2^k-1]$ and U be random variables and let $U = U' + 2^k \sum_{i=1}^{n} U'_i$, where $U' \in_R [0..2^k-1]$ and each U'_i is uniform and independent in $[0..2^\kappa-1]$. Then:*

$$\Delta(X+U;U) < 2^{-\kappa}. \tag{5}$$

Proof. Let $U_n = U' + 2^k U'_n$ and $U_i = 2^k U'_i$ for $i = 1, \ldots, n-1$. Observe that U_n is uniform in $[0..2^{k+\kappa}-1]$, and U_i are independent. Also, let

$$f : [0..(n-1)2^k(2^\kappa-1)] \times [0..2^k(1+2^\kappa)-2] \to [0..2^k(2+n(2^\kappa-1))-2]$$

be defined as

$$f(x,y) := x + y$$

Using the same method as in the proof of Theorem 1 we obtain:

$$\begin{aligned}
\Delta(X+U;U) &= \Delta(X+\sum_{i=1}^{n} U_i; \sum_{i=1}^{n} U_i) = \Delta(X+\sum_{i=1}^{n-1} U_i + U_n; \sum_{i=1}^{n-1} U_i + U_n) \\
&= \Delta(f(\sum_{i=1}^{n-1} U_i, X+U_n); f(\sum_{i=1}^{n-1} U_i, U_n)) \\
&\text{Lemma 2} \leq \Delta((\sum_{i=1}^{n-1} U_i, X+U_n); (\sum_{i=1}^{n-1} U_i, U_n)) \\
&\text{Lemma 3} = \Delta(X+U_n; U_n) \\
&= \Delta(X+U'+2^k U'_n; U'+2^k U'_n) \\
&\text{Lemma 1} \leq \frac{2^k-1}{2^{k+\kappa}} < 2^{-\kappa}.
\end{aligned}$$